Graphics rendering pipeline: Shaders for lighting effects

Autor: Juan Vicente Guillén Casas

Director/a: Ana Gil Luezas

Curso 2018/2019

Trabajo de fin de grado del Grado en Doble Grado en Ingeniería Informática - Matemáticas, Universidad Complutense de Madrid

Fecha: 20 de Septiembre del 2019

Autorización de difusión

Juan Vicente Guillén Casas

20 de Septiembre de 2019

El/la abajo firmante, matriculado/a en el Doble Grado en Ingeniería Informática y Matemáticas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado "Graphics rendering pipeline: Shaders for lighting effects", realizado durante el curso académico 2018-2019 bajo la dirección de Ana Gil Luezas en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Juan Vicente Guillén Casas

Resumen

La simulación del comportamiento de la luz en tiempo real en escenarios tridimensionales es cada vez más realista gracias a la evolución del hardware gráfico (GPUs) que va incrementando el rendimiento y ofreciendo nuevas características de programación.

Los efectos visuales que produce la luz sobre una escena 3D se pueden abordar desde la perspectiva de los principios físicos de la luz, con modelos de renderizado específicos, como el cálculo de las ecuaciones de reflectancia y refracción; o introduciendo elementos de realismo a partir del modelo básico más utilizado en la tubería de renderizado, mejorando la descripción de los materiales de los objetos añadiendo a las texturas de reflexión difusa y especular, datos para detallar las rugosidades del material, mejorando los brillos y añadiendo generación de sombras.

En este trabajo se estudian las técnicas para la mejora de los efectos lumínicos en el modelo de renderizado basado en la tubería gráfica, se implementan utilizando las tecnologías más actuales de programación gráfica sobre GPU y se integran en una aplicación que permite renderizar escenas 3D cargadas de archivo.

Palabras clave

GPU, Shader, OpenGL, Sombras, Efectos lumínicos, Escena 3D.

Abstract

Simulation of real-time light behavior in three-dimensional scenarios is increasingly realistic thanks to the evolution of graphic hardware (GPU) that improves its performance and incorporates new programming features.

Lighting effects can be aproached by specific visualization models based on the calculation of the reflectance and refraction equations, or by improving the basic model of the graphics pipeline by adding new features to the material of the 3D models.

In this work we study techniques to improve the effects of lighting in the graphics rendering pipeline, they are implemented using the most current graphic programming technologies in GPU, and they are integrated into an application that allows to render 3D scenes loaded from a file.

Keywords

GPU, Shader, OpenGL, Shadows, Lighting effects, 3D scene.

Índice

1. Introducción

Empezamos este trabajo de fin de grado dando una pequeña introducción básica a todo lo que vamos a abordar. Todos nos hemos topado alguna vez con un videojuego o una animación 3D de una película, lo que no todos se preguntan es cómo están programados todos sus colores, sus formas, sus sombras, etc. Esto por supuesto requiere del trabajo de los programadores que se encargaron de programar dichas animaciones para que una tarjeta gráfica se encargara de ejecutarlas y prepararlas para que una pantalla las pueda mostrar al espectador.

Nuestro proyecto va a tratar de introducirnos al mundo de estos programadores, conocer el entorno donde se programan todas estas animaciones y aprender a hacer lo más básico como puede ser dibujar una figura en 3D y aplicarle las texturas, la iluminación y el sombreado correctamente para que parezca lo más real posible.

1.1. Antecedentes, Objetivos y Plan de este trabajo

En este apartado vamos a abordar los antecedentes previos a la realización del proyecto, los objetivos que tiene en mente dicho trabajo y el plan de trabajo que hemos llevado a cabo, desde investigar sobre la materia hasta poner en práctica los conocimientos adquiridos en el proceso.

1.1.1. Antecedentes a este trabajo

En esta parte vamos a tener que hablar de la base sobre la que partimos previamente antes de ponernos en materia. Debemos aclarar que no hemos tenido en nuestro plan de estudios ninguna asignatura relacionada con la programación gráfica, por lo que partimos desde cero prácticamente. Lo cual nos obliga a tener una parte de investigación sobre como funciona una tarjeta gráfica, la tubería de renderizado, la programación de ésta con shaders y los distintos efectos gráficos que se pueden realizar dando uso de éstos.

Una vez tengamos una base previa asentada por esta investigación, podremos asentar estas bases como conocidas y en este proyecto las daremos como tal, ya hablaremos de esto en una sección posterior. Con esta previa podremos empezar a desarrollar nuestro trabajo e investigar más a fondo los efectos gráficos que queremos exponer en nuestro proyecto.

1.1.2. Objetivos del trabajo

El principal objetivo de este proyecto es completar mi aprendizaje en este área de conocimiento que no pude aprender a lo largo de la carrera, que a la vez es muy importante y utilizado en el mundo laboral. Este aprendizaje quiero que dé producto a la realización de un pequeño motor gráfico que sea capar de recrear escenas sencillas con figuras básicas

que dispongan de texturas capaces de recrear profundidades, también de generar una iluminación, ya que el trabajo tendrá un gran hincapié en ésta debido a que es la encargada de que la mayor parte de cosas que vamos a desarrollar se puedan recrear correctamente, como por ejemplo los efectos surgidos de ésta, como las sombras, los efectos de transparencia y la floración de un foco emisor de luz. También trataremos de hacer que renderice pequeños sistemas de partículas para dar un efecto de animación a nuestra escena. Y por último para aproximarnos al mundo de los videojuegos, que pueda soportar la carga de modelos externos diseñados por artistas.

Otra función que tiene este trabajo es servir como guía de aprendizaje para aquellos que tengan un conocimiento muy básico sobre la materia y quieran ampliarlo más, que al leer y comprender los apartados de este trabajo les abra un gran interés sobre el uso de los shader a la hora de generación de gráficos y a la vez aprendan a utilizar los que van a ser descritos en este trabajo.

1.1.3. Plan de Trabajo

Para la realización correcta del trabajo tendremos que organizar distintas etapas, estas etapas deben de estar marcadas por el patrón de Aprendizaje-Realización-Resultados. Para ello hemos decidido dividirlo en las siguientes etapas:

- **Etapa de investigación**: En esta etapa nos ocuparemos de aprender sobre el funcionamiento básico de la programación gráfica, tubería de renderizado, shaders y buffers correspondientes. Una vez podamos entender como funcionan éstos perfectamente, tendremos un proceso de selección de los distintos efectos gráficos que queremos llevar a cabo y una vez elegidos, tendremos que investigar como se programan los shaders para que se puedan realizar tales efectos. Esta es la fase que más tiempo nos llevará debido a nuestra inexperiencia con esta parte de la programación.

- **Etapa de redacción y programación**: En esta etapa ya habremos aprendido todos los conceptos que vamos a introducir en nuestro proyecto, tendremos que llevar dos actividades en paralelo. La primera consiste en la redacción de esta memoria, exponiendo los conceptos tecnológicos previos para introducir al lector a los entornos de programación en los que vamos a trabajar, una introducción al Álgebra Lineal para que podamos entender las propiedades de las aplicaciones lineales y la importancia de éstas en las operaciones de nuestros shaders y finalmente exponer cada efecto gráfico y los procedimientos que llevamos a cabo para poder realizarlos.

  La segunda actividad que se va a desarrollar a la vez es la programación de nuestro motor gráfico que se va a encargar de llevar a la pantalla los distintos efectos que vamos a tener en nuestro trabajo. En la memoria deberemos explicar todo el funcionamiento de nuestro programa, ya sean todas sus clases y las funciones que desempeñan cada una, así como los shaders que utilizaremos a la hora de trabajar con nuestra GPU.

- **Etapa de ejecución y obtención de resultados**: En esta última etapa deberemos pensar en diferentes escenarios para llevar a cabo ciertas pruebas que verificarán el funcionamiento correcto de nuestro programa, una vez terminen todas satisfactoriamente habremos terminado nuestro proyecto. Acabaremos la memoria con una breve conclusión final a raiz de los resultados obtenidos en nuestras pruebas.

1.2. ¿Qué es OpenGL?

Como hemos explicado en la sección anterior, nuestro objetivo es aprender y comprender cómo se genera y ejecuta el código en nuestra tarjeta gráfica para poder generar los gráficos. Por lo cual necesitaremos de una herramienta para poder pasar nuestro código a la tarjeta, debemos de tener en cuenta que cada tarjeta requiere de una arquitectura diferente, aquí es donde entra en juego OpenGl que no es más que una API (En realidad se considera una especificación que siguen API encargadas de la generación de gráficos) que se dedica a generalizar el paso de nuestro código a cualquier tarjeta gráfica [0].

La estructura que nos aporte esta API será un modelo de generación de gráficos a partir de primitivas, en nuestro caso usaremos lineas y triángulos, en forma de máquina de estados, es decir, las librerías que implementen de ésta, funcionarán almacenando variables que determinarán cómo se van a dibujar nuestros gráficos.

Existen muchas librerías que se implementan para funcionar con OpenGL y además extenderse con algunas funcionalidades que facilitan al usuario trabajar con éste, de las cuales podemos destacar GLUT, SDL y por último y la que vamos a utilizar para la realización de nuestro proyecto, GLFW, la cual es en mi opinión la más adecuada para introducirnos por primera vez en éste área ya que nos aporta la capacidad de crear ventanas que acepten inputs que sean completamente compatibles con la especificación de OpenGL.

1.2.1. Herramientas y librerías que vamos a usar con OpenGL

En esta sección vamos a detallar todas las herramientas o librerías que nos van a servir para poder generar nuestro primer programa gráfico, nuestro objetivo es poder explicar en qué consisten y el motivo por el cual las utilizamos.

Empezaremos por GLFW, que ya la hemos mencionado anteriormente:

- **GLFW:**

  GLFW es una librería dirigida a openGL, su función pasa principalmente por la generación de ventanas compatibles con nuestra especificación en las que se van a proyectar todos los gráficos que rendericemos. También cubrirá la gestión de los inputs, ya sean por teclado como de movimiento de ratón [1].

En nuestro caso nos va a ayudar a poder controlar nuestra aplicación, pudiendo configurar ciertos parámetros con el teclado o ratón a nuestro antojo, como por ejemplo para poder mover nuestra cámara en 3D como queramos. También usaremos los estados de las ventanas para poder finalizar la ejecución de nuestro programa.

Una vez visto el funcionamiento básico de esta librería vamos a introducirnos al siguiente apartado. Como bien hemos dicho al principio de esta sección, openGL es una especificación para generalizar la generación de gráficos y aplicaciones 3D en la cual se tienen definidas una gran cantidad de funciones o métodos para la gestión de éstos. En este contexto abrimos paso a la siguiente librería.

- **GLAD:**

Como también es de saber, existen una gran variedad de tarjetas gráficas que requieren del uso de unas funciones y prescindir de otras, es más requieren de unas posiciones específicas de la memoria que no se pueden saber en tiempo de compilación, por lo que se tienen que asignar a tiempo de ejecución, y ese trabajo es demasiado pesado para tener que hacerlo usando las engorrosas direcciones y métodos definidos en openGL. Por lo que GLAD se encarga de facilitarnos esa tarea, tenemos una librería encargada de cargar en nuestro programa todas las direcciones de las variables y funciones que nuestra versión de openGL necesita para poder trabajar en nuestra tarjeta gráfica correctamente [2].

Gracias a esta herramienta tendremos cargadas todas las funciones y variables de nuestro openGL y no tendremos que ir cargando una por una cada vez que necesitemos utilizarla para alguna función en especifico. Para poder obtener el cargador adecuado de GLAD para nuestra versión solo tendremos que acceder a la página oficial, introducir el openGL que queremos ejecutar y nos generara el archivo glad correspondiente para poder cargar sus funciones y variables correspondientes.

Para la siguiente parte nos adentraremos ya en las herramientas que gestionan las operaciones internas para generar los gráficos y poder mostrarlos en pantalla. Aquí introducimos una herramienta fundamental que va a ser el lenguaje en el que van a ir programados nuestros shaders que van a calcular los vértices y los fragmentos que van a componer a nuestras figuras.

- **GLSL:**

  Conocemos a GLSL como un lenguaje, básicamente similar a C, que es utilizado para poder codificar los shaders. Este lenguaje destaca por su facilidad y eficiencia a la hora de hacer operaciones sobre vectores, obviamente debido a que los shaders trabajan con vectores para poder generar las figuras en un espacio. Además de ser un código que gestiona bastante bien los programas de entrada y salida de variables que es básicamente el funcionamiento que debe de tener un shader, podemos ver más a fondo este lenguaje en su página oficial [3]. Para que podamos emplear todas estas operaciones necesitamos ayudarnos de alguna biblioteca que contenga lo que necesitamos, por suerte OpenGL dispone de librerías para poder realizar las operaciones sobre vectores, de las cuales vamos a destacar GLM encargada de implementar una amplia gama de aplicaciones lineales que nos ayudarán a la hora de calcular la posición en la que dibujar nuestras figuras.

- **GLM:**

  Como bien hemos mencionado, GLM es una biblioteca/ librería basada en las espacificaciones de GLSL [4], que es capaz de implementar aplicaciones lineales como por ejemplo traslaciones, rotaciones, proyecciones, etc. Además de poseer operaciones sobre vectores para diversos usos, como por ejemplo poder normalizar vectores, es decir hacerlos unitarios, que nos vendrá bastante bien para hacer operaciones con la iluminación. También podemos destacar la aportación de los productos: escalar y vectorial que nos dan un gran avance para poder calcular vectores importantes a la hora de generar vértices en un shader geométrico o conseguir propiedades para aplicar en el shader de fragmentos.

- **Asimp:**

  Esta herramienta no la vamos a abordar mucho en esta sección puesto que nos vamos a topar con ella más adelante, aunque podemos adelantar que nos va a servir para poder cargar los modelos de mallas y prepararlos para que nuestros shaders puedan operar con ellos y así generar en nuestra pantalla estos maravillosos modelos diseñados por profesionales del modelaje 3D.

1.3. ¿Qué conceptos vamos a asumir que están conocidos?

OpenGL abarca una gran cantidad de conocimientos introductorios por lo que el lector deberá dar por conocidos ciertos conocimientos que vamos a resumir en este apartado:

Para empezar, debemos conocer que los vértices vienen cargados en un buffer conocido como VBO (Vertex Buffer Object) el cual contendrá todos los detalles de los vértices, desde la posición, colores hasta las normales y tangentes que éste aporta, que serán cargados posteriormente en el shader. También tenemos que saber que en el buffer los vértices vendrán cargados uno detrás del otro y con los detalles escritos de forma consecutiva, pongamos un ejemplo, si un vértice tiene posición, color y normal, éstas se escribirán de manera consecutiva.

Para poder apuntar a estos detalles de cada vértice necesitamos otro buffer que apunte directamente a cada uno, conocido como el VAO (Vertex array object) es decir que si queremos saber el color de un vértice, que se escribe en el VBO justo después de la posición, tendremos que acceder a la posición segunda del VAO que nos asegurará de manera mucho más rápida que lo que estamos mirando son colores. Daremos por conocido como cargar un VAO en un programa para vincularlo a un VBO para que apunte a sus atributos, si no podemos consultar en la página oficial [5].

Por último, para poder determinar en que orden vamos a leer nuestros vértices, ahorrándonos tener que escribir vértices de sobra cuando no es necesario, para esto tenemos el buffer de índices que nos indicará en orden en el que se tienen que ir leyendo los vértices de nuestro buffer de vértices. Tendremos que también vincularlo al VAO junto al VBO para que pueda tomar efecto y facilitarnos la vida.

Otro detalle de OpenGL que esperamos conocer previamente son las Texturas y como cargarlas en nuestro buffer para que el shader de fragmentos se encargue de dibujarlas. Así como configurar las propiedades de nuestra/s textura/s previamente antes de que el shader las dibuje. Conocer el manejo de una textura es muy importante para poder conocer las técnicas que vamos a utilizar para poder generar las propiedades gráficas que va a abordar nuestro trabajo.

Cabe también recordar que las texturas una vez se configuren sus propiedades, se va a cargar en nuestro VBO las coordenadas en las que se desea dibujar nuestra textura y a la vez la imagen de nuestra textura va a ser cargada en una variable uniforme en el shader de fragmentos, en dicho shader se va a dibujar dicha imagen en la posición que indiquen tales coordenadas [6].

Una vez repasados estos conceptos muy básicos estamos preparados para poder comprender los siguientes apartados, ya que a partir de ahora ésto se va a dar como completamente conocido por el lector, en la siguiente parte vamos a hablar de la parte más importante de openGL y los encargados de hacer que los gráficos se puedan calcular y generar en nuestra tarjeta, los shaders.

1.4. ¿Qué son los Shaders y como funcionan en nuestra tarjeta gráfica?

Nuestra tarjeta no puede recibir el código tal cual, necesita de una adaptación previa para poder recibir los vértices, texturas, etc, y poder realizar las operaciones sobre estas. Es aquí donde aparecen los shaders, que son los programas que nosotros escribimos para que ejecuten en la tarjeta dichas operaciones. Por lo que son lo más importante a la hora de generar nuestras figuras, ya que es donde realmente se está construyendo todo para poder ser proyectado en nuestra pantalla.

Para poder generar bien nuestros gráficos necesitamos de una serie de procedimientos para gestionar los vértices, texturas y otros parámetros que nos vienen como entrada para que finalmente acabe saliendo el producto que todos esperamos, entre esos procedimientos vamos a distinguir los tipos de shaders que vamos a abordar en esta sección:

1.4.1. Shaders de vértices

El primer paso que se va a realizar cuando empieza a trabajar nuestra GPU es calcular la localización real de nuestros vértices, vectores normales, coordenadas de texturas, y toda componente que necesite ser reubicada en el espacio. El shader de vértices es aquel que se va a encargar de dicha tarea, bastante importante a la hora de trabajar enl 3D como ya veremos más adelante, vamos a tener que cargarle previamente muchas aplicaciones lineales, ya que son las encargadas de mover nuestros puntos en el espacio, podemos consultar más sobre el shader de vertices aquí [7].

Las variables de salida van a ser los puntos en el espacio que van a ser procesados para generar la primitiva que tengamos configurada, en nuestro caso van a ser triángulos, y una vez generadas dichas primitivas tenemos dos opciones, que la primitiva pase por el shader geométrico para añadir o quitar vértices nuevos y así generar otras figuras, o bien pasar directamente al procedimiento de rasterización, donde se generan los fragmentos de nuestra figura que serán la entrada al shader de fragmentos. En la siguiente imagen podemos apreciar un ejemplo de shader de vértices:

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos, 1.0);
}
```

Figura 1: Ejemplo sencillo de un shader de vértices.

1.4.2. Shaders de fragmentos

El último procedimiento que se realiza en nuestra GPU es generar la figura dibujandola a base de fragmentos, que podemos considerar pequeñas cuadrículas dibujadas a lo largo de la primitiva que se ha dibujado en el anterior procedimiento. El shader de fragmentos es el que se encarga de tratar estas cuadrículas para que sean posteriormente dibujadas en nuestra pantalla ya sea pintandoles el color, como calculando su grado de iluminación así como la sombra que estos reciben de otros objetos que se están también renderizando, podemos consultar más sobre el shader de fragmentos aquí [8].

El shader de fragmentos se va a ejecutar en la última parte de todo el proceso por lo que los resultados obtenidos en esta parte van a ser los que básicamente se dibujen en pantalla. También tenemos que destacar la gran importancia de este proceso a la hora de recrear efectos lumínicos ya que va a ser en esta etapa cuando calculemos todas las propiedades como la reflexión, intensidad, sombreado, profundidad de las texturas, brillo, etc. Aquí podemos ver un ejemplo sencillo de shader de fragmentos:

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(vec3(0.5), 1.0);
}
```

Figura 2: Ejemplo sencillo de un shader de fragmentos.

1.4.3. Shaders geométricos

Este shader se corresponde con el paso intermedio entre los dos procedimientos vistos anteriormente, su objetivo principal es eliminar o generar nuevos vértices y primitivas según se requiera, para adecuar estos a cada frame de renderizado sin necesidad de realizarlo en la CPU. Este shader es el más reciente de los tres, pero no es obligatorio. Sus aplicaciones son muy variadas, en este trabajo lo utilizamos para implementar un sistema de partículas y para la generación de sombras con una textura cúbica, podemos ver más sobre shaders geométricos aquí [9].

Este shader también requiere de que el lector conozca muy bien el funcionamiento de los anteriores y de los procesos que hay entre medias de la ejecución de cada shader, debido a que debemos de saber que en el shader de vértices las posiciones en el espacio de los vértices van a ser calculadas y una vez salgan va a haber un procedimiento que pinte con ellos una primitiva

indicada, esta primitiva va a ser la que entre en nuestro shader geométrico y deberemos de saber como acceder a las posiciones de los vértices, como generar nuevas primitivas y con ellas nuevas figuras (Dando uso de las funciones EmitVertex y EmitPrimitive), sabiendo el uso de ésto la figura que saldrá de nuestro pequeño programa será la que será tratada para generar los fragmentos en el siguiente procedimiento. En la siguiente imagen podemos ver un ejemplo sencillo de shader geométrico:

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Figura 3: Ejemplo sencillo de un shader geométrico.

1.5. Recreación de modelos 3D con mallas

En esta última parte de la introducción vamos a hablar sobre una herramienta muy importante a la hora de cargar modelos externos a nuestra escena, tenemos que tener en cuenta que no todos los gráficos que generamos están hechos desde el principio por nuestra parte, que gran parte tendremos que cargarlos de archivos exportados por aplicaciones de diseño gráfico especializadas en diseñar este tipo de modelos, en esta ocasión utilizaremos Blender aunque nos descargaremos modelos ya terminados debido a que no somos profesionales del diseño 3D.

En nuestro caso vamos a utilizar una librería especializada en cargar archivos de modelos 3D conocida como Assimp, que va a transformar los datos del archivo en una estructura adaptada para poder ser renderizada en nuestros shaders y ser puesta en escena. Empezaremos describiendo dos conceptos previos para poder entender todo el funcionamiento de Assimp:

1.5.1. ¿Qué es una Malla?

Conocemos por malla al conjunto de datos que definen a un componente 3D, ya sea las coordenadas de sus vértices, sus vectores normales, sus coordenadas para texturas y los materiales de los que va a disponer, donde destacamos en cada material las texturas que se van a dibujar en la malla y características para los tipos de luz que van a iluminar al objeto (Se explicará más adelante en los shaders de iluminación). En otras palabras, la malla define lo que va a ser el componente una vez se renderice en la pantalla de nuestro ordenador. En nuestro caso usaremos una clase que represente a un objeto de tipo Malla con estas características [10].

Normalmente en el diseño 3D los objetos suelen estar compuestos de varias mallas que van generando distintas capas del objeto, cada malla renderizará su parte hasta llegar al producto final. Esto es comúnmente conocido como Modelo que es la composición de varias mallas para así formar un objeto final.

1.5.2. ¿Qué es un Modelo?

Como bien decimos antes, un modelo está compuesto por una o más mallas que se encargan de definir sus distintas partes, en este caso será el encargado de cargar las texturas en cada malla según proceda y también el encargado de renderizar y cargar todos los componentes de cada malla según el orden que proceda. Diremos que es la parte en la que podremos controlar la generación de nuestro objeto 3D, utilizaremos una clase que represente a este tipo de modelos que adaptaremos y le realizaremos unas ligeras modificaciones para adaptarlo a nuestro motor, podemos consultar aquí [11].

Después de haber descrito estas dos partes fundamentales de la estructura de un modelo gráfico 3D, nos disponemos a explicar el funcionamiento de Assimp y cómo lo vamos a utilizar en nuestro proyecto para poder cargar modelos más realistas para el motor gráfico.

1.5.3. Repasando los detalles de Assimp

Ahora que hemos visto qué son las Mallas y los Modelos podemos introducirnos de lleno a Assimp, como bien hemos dicho antes, es una librería encargada de convertir un diseño 3D de un archivo .obj generado por una aplicación de diseño 3D a una estructura de datos lista para poder ser renderizada y puesta en pantalla, esta estructura cargará los datos en los buffers correspondientes para que posteriormente actúen los shaders sobre ellos.

La librería cargará el modelo en una clase objeto llamada Scene que tendrá como atributos un nodo raíz que apuntará a los nodos hijos, un vector que contendrá todas las mallas del modelo y otro que contendrá todas las texturas que se usen en todo el modelo. Como hemos mencionado, existe un nodo raíz, es decir, que la escena estará compuesta por nodos que apuntarán a sus hijos y contendrán los índices que apunten a las mallas que se corresponden

con dicho nodo. Esta estructura arbórea nos ayuda a generar una escena con un cierto orden y estructura de renderización con la que podremos darle juego para renderizar el modelo final.

Como es de esperar, el objeto que contendrá todas las propiedades de las mallas tendrá el mismo nombre y consistirá en nuestro caso de: Un vector con las coordenadas de los vértices, otro con las coordenadas de texturas y otro con las normales, además de un vector que contenga los índices para dibujar las primitivas en el orden correcto, en caso de tener más de una primitiva tendremos que agrupar estos vectores por cada figura pero en nuestro caso solo usamos triangulos por lo que solo tendremos un vector con los indices correspondientes. Y por último, también tendrá que tener unos punteros que apunten a las texturas cargadas en el objeto Scene, una vez con todo esto, tendremos una malla lista para poder renderizar. Podemos consultar más sobre Assimp aquí [12] y [13].

Ahora ya que hemos visto cómo funciona la librería de Assimp podemos entender cómo vamos a poder cargar modelos complejos en nuestro motor para poder mejorar la calidad de las escenas que éste va a poder renderizar, además de que al tener todas las propiedades bien cargadas nos será más fácil aplicar nuestros shaders de sombreado, iluminación, profundidad de texturas, etc.

1.5.4. Cargar un modelo en nuestro programa

Una vez visto Assimp tendremos que ponerlo en marcha, necesitaremos crear un directorio donde poder cargar los .obj de nuestros modelos (.obj es uno de los formatos en los que trabaja Assimp para poder cargar los modelos), ahora solo tenemos que crear el objeto Scene y cargarle del directorio los parámetros que necesite, ya sean vértices, normales, texturas, etc. Este se encargará de crear los objetos de malla y cargarles todos los parámetros así como crear las texturas para que éstos puedan cargarlas. Además deberemos cargar en nuestros shaders todos los parámetros necesarios de Scene para que éstos puedan realizar sus operaciones correctamente y así generar nuestro modelo en pantalla.

Visto esto ya conocemos todo lo básico respecto a OpenGL para poder adentrarnos en el contenido principal del trabajo, los diferentes tipos de Shaders para generar distintos efectos gráficos, ahora solo nos queda ver la base matemática que sustenta el funcionamiento correcto de éstos.

2. Matemáticas utilizadas para el desarrollo de nuestros Shaders

En esta sección del proyecto toca hablar de conceptos que tocan la rama de las Matemáticas que vamos a tener que conocer antes de adentrarnos en la explicación de los shaders, intentaremos explicarlo aplicando nuestro conocimiento adquirido a lo largo de la carrera.

Tenemos que tener en cuenta que este proyecto tiene un porcentaje de contenido en Álgebra Lineal, ya que es aplicar transformaciones lineales para ubicar nuestro punto en un espacio global, moverlo hacia nuestra cámara y prepararlo para que pueda ser proyectado en nuestra pantalla.

2.1. Introducción a las aplicaciones lineales

Empezaremos con lo más básico para poder introducirnos a las aplicaciones lineales más importantes que se usan en nuestros shaders, empezaremos definiendo qué es una aplicación lineal y qué propiedades tienen [14]:

- **(Definición)**
  Sean W y V dos espacios vectoriales en un cuerpo $\mathbb{K}$, Diremos que $f : W \longrightarrow V$ es una aplicación lineal si:
  $f(\nu * u + \alpha * v) = \nu * f(u) + \alpha * f(v)$ $\forall u, v \in W$ y $\forall \nu, \alpha \in \mathbb{K}$

- **(Definición)**
  Sea f una aplicación lineal diremos que es un monomorfismo si es inyectiva, epimorfismo si es sobreyectiva e isomorfismo si es biyectiva. Si el conjunto de partida y salida es el mismo en vez de isomorfismo lo conoceremos como automorfismo.

Una vez vistas la definiciones más básicas, damos a conocer las propiedades más importantes de las aplicaciones lineales para así poder visualizarlas de una mejor manera, destacamos las siguientes:

- **(Propiedad)**
  Sea f una aplicación lineal, entonces f es un monomorfismo $\iff ker(f) = 0$, dónde ker(f) es el espacio anulador de f.

- **(Propiedad)**
  Sea $f : W \longrightarrow V$ una aplicación lineal, entonces $dim(Im(f)) \leq dim(W)$, dónde Im(f) es el espacio imagen de f.

- **(Propiedad)**
  Sea $\{u_1, ...; u_n\}$ una base de W y $\{v_1, ...; v_n\}$ un conjunto de vectores cualesquiera de V, entonces existe una única aplicación lineal f, tal que:
  $f(u_i) = v_i$, $\forall i \in \{1, ..., n\}$.

- **(Propiedad)**
  Toda aplicación lineal de un espacio W de dimensión m hacia otro espacio V, de dimensión n, puede ser representada por una matriz A de dimensión $n \times m$. De forma que sea un punto $x \in W$, y su imagen $y = f(x) \in$ V entonces: $y = A * x$

- **(Propiedad)**
  Sea una aplicación lineal $f : W \longrightarrow V$, con A matriz que la representa, entonces:
  - $dim(Ker(f)) + dim(Im(f)) = dim(W)$.
  - $dim(Im(f)) = rg(A)$

- **(Propiedad)**
  La composición de dos aplicaciones lineales es también lineal.

  Y por último veamos los cambios de base y cómo afectan éstos a las aplicaciones lineales para completar la introducción de éstas, empezamos hablando de la matriz de cambio de base respecto a las coordenadas:

- **(Propiedad)**
  Sean W y V dos espacios vectoriales, siendo $\{w_1, ..., w_n\}$ $\{v_1, ..., v_n\}$ sus bases respectivamente. Teniendo que para una cierta matriz regular P:

  $(v_1, ..., v_n) = (w_1, ..., w_n) * P$

  Entonces, tenemos que siendo X_w las coordenadas de un punto en W y X_v las coordenadas de su equivalente en V, tenemos que P es la matriz de cambio de base, por lo que: $X_w =$ P * X_v .

  Ahora finalizamos esta introducción añadiendo la influencia de las matrices de cambio de base en una aplicación lineal:

- **(Propiedad)**
  Sea $f : W \longrightarrow V$ una aplicación lineal, con B_1, B_2 bases de W y B_3, B_4 bases de V, A_{B_1,B_3} matriz de f con B_1, B_3 bases de W, V respectivamente y A_{B_2,B_4} lo mismo pero con las otras bases restantes, entonces tendremos lo siguiente:
  - $A_{B_2,B_4} = Q^{-1} * A_{B_1,B_3} * P$ dónde Q es la matriz de cambio de base entre B_3 y B_4, y P es la del cambio entre B_1 y B_2, es decir: $B_2 = B_1 * P$ y $B_4 = B_3 * Q$.
  - Si $W = V$ y también tenemos que $B_1 = B_3$ y $B_2 = B_4$ entonces: A_{B_1,B_3} y A_{B_2,B_4} son semejantes, es decir que existe P regular tal que: $A_{B_2,B_4} = P^{-1} * A_{B_1,B_3} * P$.

Una vez vistas las propiedades más básicas estamos listos para poder conocer las aplicaciones lineales que vamos a necesitar en nuestro proyecto, como vamos a trabajar sobre un

espacio en 3 dimensiones con un sistema de coordenadas homogéneo, tenemos que definir las aplicaciones lineales en un espacio en $\mathbb{R}_4$.

2.2. Traslación

Esta aplicación lineal se encargará de mover los puntos del espacio en una dirección concreta [15]. Tratamos con una matriz del estilo:

$$\mathrm{T} = \begin{pmatrix} 1 & 0 & 0 & t_0 \\ 0 & 1 & 0 & t_1 \\ 0 & 0 & 1 & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$, con (t_0, t_1, t_2) el denominado vector de traslación que será

la dirección en la que moverá la aplicación los puntos.

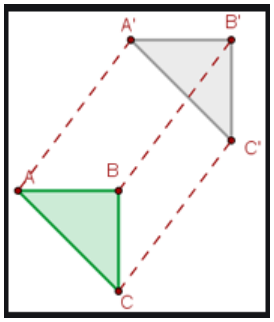


Figura 4: Traslación de una figura.

(Propiedad) La inversa de la traslación es bastante similar, solo tendremos poner negativo su vector de traslación:

$$T^{-1} = \begin{pmatrix} 1 & 0 & 0 & -t_0 \\ 0 & 1 & 0 & -t_1 \\ 0 & 0 & 1 & -t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esto tiene bastante sentido ya que lo que estamos haciendo es devolver el punto a su posición original, por lo que tenemos que usar la misma dirección pero negada.

2.3. Rotación

Esta aplicación se va a encargar de girar, sobre un eje determinado como referencia, los puntos del espacio [16]. Luego necesitamos conocer dos cosas antes de poder mostrar la matriz, la primera es el eje sobre el que vamos a rotar nuestro espacio, y la segunda es el ángulo

que va a determinar cuanto rotamos. Dado un ángulo θ tenemos las siguientes aplicaciones de rotación en los ejes X, Y y Z.

Eje X : $$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Eje Y: $$R = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Eje Z: $$R = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

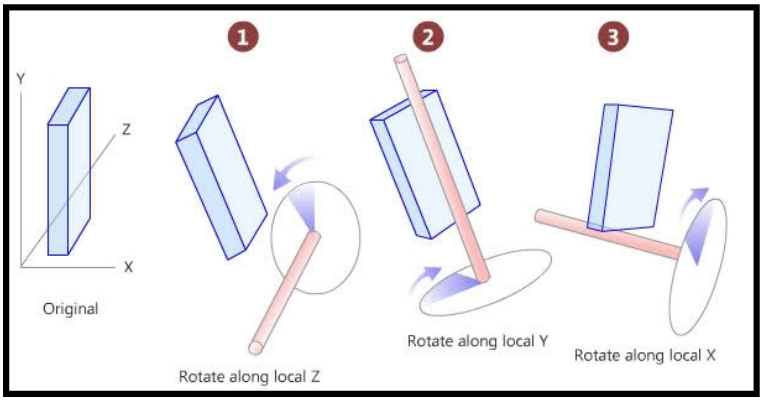


Figura 5: Rotación de una figura en cada uno de los 3 ejes.

2.4. Cambio de escala

Para el cambio de escala, tenemos que pensar en si queremos reducir o aumentar el objeto pese a que se puedan mover sus vértices o si queremos preservar algunos [15]. En el primer caso tenemos las siguiente matriz para nuestra aplicación:

$$S = \begin{pmatrix} s_0 & 0 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$, con (s_0, s_1, s_2) el vector de reducción que se encargará

de reducir el objeto según indiquemos en cada componente. Si las componentes son iguales le llamaremos escala uniforme, y no uniforme en el otro caso.

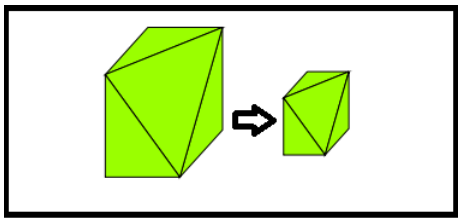


Figura 6: Aplicando la aplicación de escala a una figura.

Ahora si queremos que al menos un vértice de nuestro objeto mantenga su posición y no se desplace necesitaremos de la siguiente transformación: Debemos de tener en cuenta la estrategia, si escalamos el vector origen nunca lo moveremos de sitio, luego tenemos que hacer una traslación previa del vértice que queramos dejar fijo, luego escalar la figura y posteriormente volver a trasladarla al sitio original con la inversa de la primera traslación.

$$S = T^{-1} * S_0 * T = \begin{pmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -x_1 \\ 0 & 0 & 1 & -x_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} s_0 & 0 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & x_1 \\ 0 & 0 & 1 & x_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} s_0 & 0 & 0 & -x_0 * s_0 + x_0 \\ 0 & s_1 & 0 & -x_1 * s_1 + x_1 \\ 0 & 0 & s_2 & -x_2 * s_2 + x_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$, con (x_0, x_1, x_2) el punto que queremos dejar

fijo y $(s_0, s_1 s_2)$ el vector de reducción.

En las siguientes dos partes vamos a hablar de las proyecciones lineales, que tratarán de desplazar las coordenadas de un espacio recortado a un espacio preparado para proyectarse en una pantalla 2D. Tenemos dos tipos de proyecciones según el tipo de recorte que tengamos en nuestro espacio, ortogonal si el recorte es un volumen rectangular y perspectiva si el volumen es piramidal (Volumen ocupado entre dos planos de distinto tamaño) [17].

2.5. Proyección perspectiva

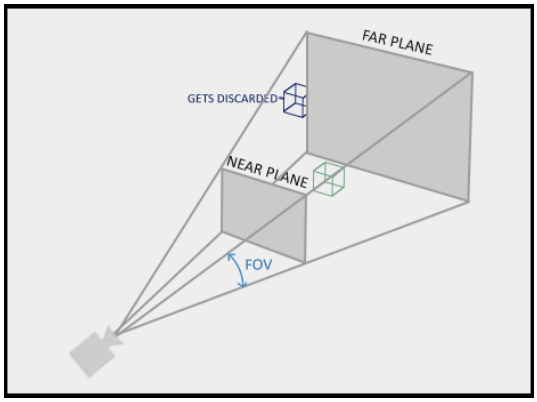


Figura 7: Proyección perspectiva.

El objetivo es transformar las coordenadas del espacio que estén dentro del espacio recortado a un espacio normalizado NDC que consistirá en el cubo $[-1,1] \times [-1,1] \times [-1,1]$. Debemos hablar antes del uso de la cuarta componente del punto de entrada (x,y,z,w), que determinará el intervalo $[-w, w]$ para transformar los puntos de entrada al espacio recortado antes de enviarlo al espacio NDC, es decir, descartar aquellos que no entren en el recorte. Para poder descartar las coordenadas que no estén en el recorte tenemos que primero proyectar en el plano más cercano a nuestro punto de referencia. Sea nuestro espacio de recorte piramidal cuyo plano cercano tiene dimensiones, $[l, r] \times [b, t]$ y cuya profundidad se define en el intervalo $[-n, -f]$, vamos a proyectar sobre el plano z = -n obteniendo como resultado:

$(x_p, y_p, z_p) = (\frac{-n*x_e}{z_e}, \frac{-n*y_e}{z_e}, -n)$

Ahora centrándonos en el objetivo principal, encontrar nuestra proyección, buscaremos una matriz P que cumpla con lo siguiente:

$$\begin{pmatrix} x_{recorte} \\ y_{recorte} \\ z_{recorte} \\ w_{recorte} \end{pmatrix} = \text{P} * \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$, además de que para un punto NDC, $$\begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} x_{recorte}/w_{recorte} \\ y_{recorte}/w_{recorte} \\ z_{recorte}/w_{recorte} \end{pmatrix}$$

Estableciendo $w_{recorte} = -z_e$ tenemos ya su cuarta fila calculada $(0, 0, -1, 0)$ el resto se calculará usando lo siguiente: Cogemos un punto en el formato NDC y lo ponemos en función de el punto proyectado del principio, es decir:

$(x_n, y_n, z_n) = (\mu * x_p + \beta, \epsilon * y_p + \beta, -n)$, donde μ, β y ϵ son parámetros que definen la relación de (x_n, y_n, z_n) con (x_p, y_p, z_p)

Sustituyendo en la ecuación del principio donde proyectamos en el plano el punto de salida obtenemos las dos primeras filas, pueden ver los cálculos aquí (Vease [17]). Pero sin embargo para la tercera tenemos un problema y es que al tener que z_p solo vale -n no podemos hacer el mismo procedimiento para calcular la tercera fila. Sabiendo que la z_c no depede de lo que valgan x e y tendremos que los dos primeros valores de la fila son 0 y además al tener $z_n = z_c/w_c = a*z_e+b*w_e/z_e$ con $w_e = 1$ ya que siempre partiremos de nuestro espacio vista. Con esto sustituyendo (z_n, z_e) por (1, -n) y por (-1,-f) obtenemos las ecuaciones que nos dan los valores a y b buscados (Vease aqui [17]). Finalmente nos queda un resultado así:

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

2.6. Proyección ortogonal

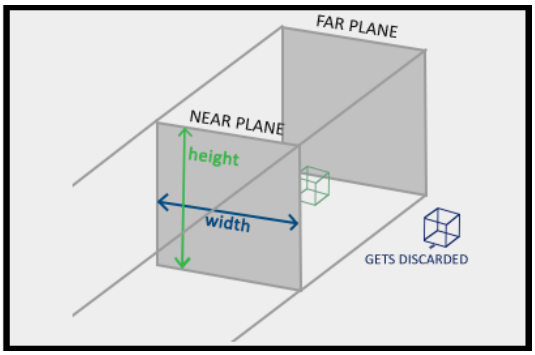


Figura 8: Proyección ortogonal.

En este caso la proyección va a ser más sencilla debido a que al tratarse de un volumen rectangular $[l, r] \times [b, t] \times [-n, -f]$ se va a poder corresponder linealmente a su formato NDC, lo cual significa que solo tendremos que hacer los siguientes cálculos para poder calcular P, calculando la correspondencia lineal de un punto NDC en función de un punto de partida (x_e, y_e, z_e) obtenemos unas ecuaciones en las que haciendo las sustituciones (x_e, x_n) por $(r, 1)$, (y_e, y_n) por $(t, 1)$ y (z_e, z_n) por $(-f, 1)$ respectivamente obtenemos:

1. $x_n = \frac{2}{r-l} * x_e - \frac{(r+l)}{r-l}$
2. $y_n = \frac{2}{t-b} * y_e - \frac{t+b}{t-b}$
3. $z_n = \frac{-2}{f-n} * z_e - \frac{f+n}{f-n}$

Además al tener que es una proyección ortogonal, la cuarta fila permanece igual en la transformación. Luego la matriz final resultante resulta ser la siguiente (Vea el procedimiento completo aquí [17]):

$$P = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{(r+l)}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.7. Cambio al espacio vista

En esta última aplicación lineal vamos a ver primero el objetivo de ésta, tenemos que trasladar el espacio global a un espacio que mueva la referencia (0,0,0) al punto de vista (x,y,z). Por lo que de primeras necesitamos una traslación que nos mueva a ese punto. Además en este espacio destino tenemos que tener como vectores base los vectores de referencia de nuestro espacio vista luego necesitaremos generar una matriz con estos que se multiplique por la traslación previa para dar así con la aplicación definitiva buscada [18].

Sean $F = (F_x, F_y, F_z)$ el vector que indica nuestro frente, U = (U_x, U_y, U_z) el que indica nuestra dirección hacia arriba y R = (R_x, R_y, R_z) el que indica nuestra derecha serán los que formarán nuestra base del nuevo espacio. Por lo que la aplicación finalmente se nos queda:

$$\mathrm{V} = \begin{pmatrix} F_x & F_y & F_z & 0 \\ U_x & U_y & U_z & 0 \\ R_x & R_y & R_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. Shaders que vamos a desarrollar en nuestro trabajo

Ahora que ya hemos visto todos los conceptos técnicos básicos y teóricos que abordan este proyecto, vamos a entrar en materia, nos introducimos a describir los efectos gráficos que vamos a introducir en nuestro pequeño motor 3D y cómo nuestros shaders van a tratar de reproducirlos. En esta parte vamos a ir describiendo cómo nuestro shader va a generar dicho efecto, ya sea los datos que recibe, cómo los procesa y la salida que éste genera para que se reproduzca en nuestras pantallas.

Vamos a empezar con los efectos más básicos y sencillos que van a aparecer en cualquier efecto que nuestro motor reproduzca, estos son la generación de un objeto 3D, cómo generar un fondo para nuestra escena con un Mapa Cúbico y cómo iluminar nuestro escenario. Empezamos por el primero que es el más importante:

3.1. Shaders para generar objetos 3D

El principal problema al que nos enfrentamos es que tenemos que representar tres dimensiones en dos que son las de nuestra pantalla, por lo que tendremos que hacer una serie de conversiones para poder llevarnos nuestro espacio a un espacio de 2 dimensiones, para ello en este procedimiento necesitaremos una serie de pasos para que proyectemos en el espacio idóneo nuestra escena y nos haga tener la sensación de estar en un ambiente 3D, de esto se encargará el shader de vértices [19].

Tenemos un total de cuatro pasos, de los cuales el último (escala y traslación al puerto de vista) se encarga openGL por lo que no le prestaremos atención. Cada paso va a transformar un espacio a otro, para llegar a un espacio que esté preparado para que openGL pueda transformarlo al puerto de vista sin ningún problema. Para ello vamos a adentrarnos en los siguientes pasos:

- **Modelado**: En este paso vamos a tratar de mover nuestro objeto, en nuestro espacio local de partida, a un espacio global donde poder posicionarlo. Para ello tendremos que coger los puntos de cada vértice y pasarlos por una aplicación lineal de modelaje que rotará, moverá y reducirá nuestro objeto para situarlo en el espacio real. En nuestro motor toda figura/objeto va a disponer de una matriz modelo que indicará su tamaño, localización en el espacio y rotación en éste.

  GLM nos aporta bastantes métodos para poder modelar nuestros objetos, de las cuales utilizaremos scale, rotate y translate para desplazar nuestra figura. Una vez pasado nuestro punto por dicha aplicación nos desplazará nuestro objeto a un espacio global donde se situará al resto de objetos cuando les toque renderizarse. Este paso deberá de hacerse en el shader de vértices ya que es el encargado de preparar los puntos para generar nuestra primitiva.

- **Visualización:** Ahora toca situarnos en dicho espacio, es decir, mover el objeto de su espacio global para que se sitúe a pie de nuestros ojos, es decir como si lo estuviéramos observando desde un punto. Para ello necesitaremos generar una aplicación lineal que mueva el espacio global a otro en el que nuestra ubicación sea el punto de referencia. Para ello tendremos que conocer 3 vectores que nos puedan hacer encontrar la matriz de transformación adecuada: Dirección frontal, derecha y arriba. Sumadas a un punto que será la referencia y que obviamente será la posición de nuestra cámara.

  Una vez que nuestros puntos pasen por dicha matriz tendremos nuestro objeto preparado para nuestra vista, en la posición en la que lo veríamos si estuviéramos dentro de ese espacio. Por lo que ahora podremos prepararlo para proyectarlo, que es el siguiente paso a realizar.

- **Proyección** En este último paso vamos a asignar nuestros puntos en un rango determinado para así poder proyectarlo al 2D en el último paso, es decir, tenemos que descartar todos aquellos puntos que se salgan fuera de ese rango y quedarnos solamente con los que vamos a ver en pantalla, para ello hay dos tipos de proyecciones: ortogonal y perspectiva, esta última va a ser la que vamos a utilizar y a explicar. Tenemos un plano cercano y otro lejano, el polígono que forman estos dos será el rango en el que podrán estar los objetos que vamos a reproducir por pantalla, el resto se descartará.

  Una vez pasados nuestros puntos por la matriz de proyección cumplirán los requisitos para que openGL pueda pasarlos a 2D y puedan ser reproducidos por pantalla.

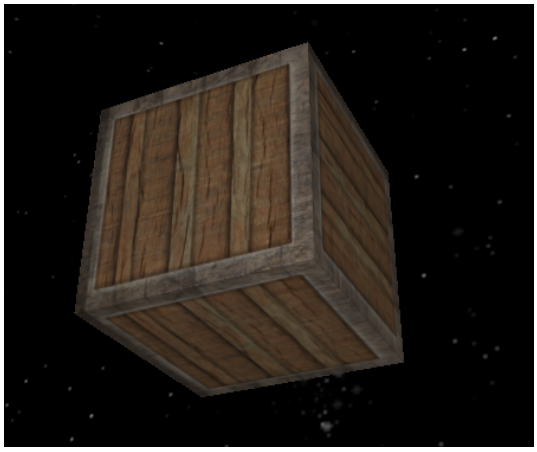


Figura 9: Objeto 3D renderizado en pantalla.

Si juntamos estas tres matrices podremos mover cualquier punto en nuestro espacio local para que tengamos el punto que va a salir en la pantalla cuando el resto de shaders hagan sus procedimientos. Cada vez que queramos reproducir una figura en 3D tendremos que realizar este procedimiento para poder ubicarla en nuestra pantalla. Los modelos de Assimp que importemos tampoco se libran de estos procedimientos. Ahora veremos como generar un fondo para hacer más realista nuestra escena.

3.2. Mapeado Cúbico y Fondo de escena

Para poder comenzar esta sección necesitamos saber que es un CubeMap y para qué se utilizan, siempre hemos hablado de texturas para poder pintar una cierta superficie, un cierto plano que mediante coordenadas de textura se va a pintar sobre ésta. Pero y si esta textura no fuera un simple plano y se tratara de un cubo, cuyas coordenadas añaden una nueva dimensión, es aquí cuando damos con la definición de un CubeMap que son texturas cúbicas que podemos utilizar para poder pintar objetos. Usaremos una función de carga

utilizada en los manuales de openGL [20], y esta nos cargará un cubeMap con las imágenes cargadas, listo para poder configurarlo.

En este proyecto vamos a utilizarlas en dos ocasiones: Para generar sombras multidireccionales y para generar un fondo de escenario. En esta sección abordaremos lo segundo y en las posteriores ya hablaremos de las sombras.

Para empezar necesitamos que el CubeMap nos haga la sensación de estar en un lugar extenso, que no podamos salirnos, por lo cual la clave para poder realizar dicho efecto está en utilizar la matriz View para modelar el fondo y que así este se aleje si vamos hacia fuera, haciendo así que nunca podamos abandonarlo y que parezca inmenso. Esto se puede hacer cogiendo la submatriz 3x3 de la matriz View y añadiéndole la cuarta dimensión para eliminar todo tipo de traslaciones, logrando dicho efecto.

Para evitar que el buffer de profundidad dibuje el fondo antes que los objetos de la escena lo ideal es primero dibujar el fondo y luego la escena pero esto a la larga puede ser bastante ineficiente ya que se renderizan pixeles que podríamos habernos ahorrado, para eso es mejor renderizarlo al final pero sabiendo que hay que engañar al buffer de profundidad para que crea que es lo más lejano que se está dibujando. Si cogemos y sustituimos en glPosition la componente z por el w que es el valor máximo con el que se puede proyectar, como ya vimos en la sección 2.5, en la etapa de proyección las coordenadas tienen un rango que es precisamente w. Con esto el buffer pensará que es lo más lejano de la escena y pintará todo lo que salga en la escena antes de que pinte el fondo.

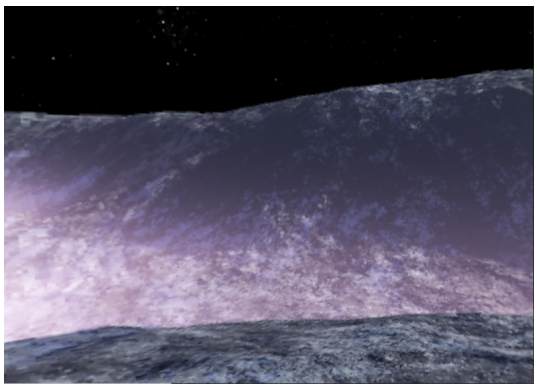


Figura 10: Fondo de pantalla en una escena.

Con esto ya solo tenemos que cargar en el shader de fragmentos, usando las coordenadas de textura, el fragmento correspondiente del fondo y ya tendríamos nuestro cubeMap con la textura de fondo completamente renderizado. Una vez visto esto vamos a ver cómo gestionar la iluminación de nuestra escena.

3.3. Shaders encargados de la iluminación

En esta parte vamos a introducirnos a la parte más importante de este proyecto que es crear efectos gráficos basados en la luz, para ello primero tenemos que saber cómo trabajan nuestros shaders para controlar la iluminación de nuestra escena, qué tipos de luces tenemos que tener en cuenta, cómo van a influir en la coloración de los fragmentos de nuestros objetos y qué vamos a mostrar en pantalla. Para ello primero deberemos de hacer una breve introducción sobre las distintas luces que vamos a tratar y los parámetros que vamos a tener en cuenta:

3.3.1. Introduciendo a los shaders de iluminación

Para poder comprender los tipos de luces que vamos a generar en este proyecto, primero tenemos que distinguir los componentes que forman la luz para poder ir más allá ya que en cada tipo de luz se calcularán de una forma algo distinta. Cabe resaltar que usaremos el conocido modelo de Phong que destaca por utilizar 3 componentes básicas para la luz: Ambiente, Difusa y Especular [21].

- **Ambiente**: Esta es la luz que el objeto va a recibir siempre, da igual los focos que le estén iluminando, esta componente siempre va a recibir un valor que se va a añadir a la luz resultante del objeto. Toda luz deberá tener un parámetro que indique el ambiente que van a recibir los objetos de ésta. Este factor se multiplicará por la textura del objeto para dar como resultado el valor lumínico respecto a esta componente.

- **Difusa**: No siempre vamos a ver un objeto de la misma forma desde cualquier ángulo de iluminación que tengamos, si la luz le viene de frente lo veremos mejor que si le viene desde otra dirección, esta componente se encarga de aportar la luz que deberíamos recibir según el ángulo en el que le llegue la luz a nuestro objeto.

  Para poder hacer este cálculo correctamente necesitaremos dos cosas, la dirección que tenga la luz respecto del punto que estemos tratando y la normal de ese punto. Esta última nos indicará el grado de inclinación que tiene el objeto. El producto escalar que nos den estos dos vectores será más grande cuanto más paralelos sean.

  Para poder modelar los vectores normales de la superficie de nuestro objeto tenemos que quedarnos únicamente con las rotaciones de la matriz modelo, por este motivo tenemos que hacerle dos operaciones a nuestra matriz para poder quedarnos con la parte 3x3 que nos interesa, por lo que la invertiremos, la traspondremos y nos quedaremos con la parte de dimensión 3 que es la que se encarga de rotar el vector. Ya podremos modelar los vectores normales a nuestro antojo.

- **Especular**: Hemos hablado antes de la luz que tiene un objeto independientemente del rayo que le venga, y dependiendo del que le venga. Pero no hemos hablado de la posición

en que estemos observando ese rayo. Es aquí cuando entra la tercera componente que se encarga de representar la intensidad en la que la luz rebota hacia nosotros. Ahora debemos tener en cuenta otro vector más, el de la dirección en la que estemos viendo el punto del objeto. Como en la Difusa también tendremos que tener en cuenta la dirección en la que nos venga la luz y la normal de ese punto.

Ahora tenemos que ver el rayo que nos rebota y para eso tenemos la función de GLM $reflect()$ que nos saca el vector complementario de la dirección de la luz respecto de la normal. El producto escalar de este vector resultante y la dirección del observador hacia el punto nos determinarán la intensidad a la que el rayo se va a reflejar. Cuanto más paralelo, más intenso nos llegará. Para poder tener en cuenta el parámetro de brillo de una superficie (No va a reflejar lo mismo el metal que la madera) elevaremos este producto a la potencia factor de brillo de éste para dar con el resultado final.

El resultado final será la suma de estas tres componentes, podemos configurar la luz de la manera que deseemos, para así poder determinar la importancia de cada componente. De ahí que en el shader existan parámetros de configuración que se multiplicarán por el resultado de cada componente para determinar la importancia de la misma, ya sea utilizando texturas difusas o especulares, valores en vectores para determinar el rango de cada componente, etc.

Una vez conocidas las componentes que tiene la luz en este modelo podemos introducirnos de lleno en los tipos de luz que trataremos según el foco de emisión que tengan. No toda la luz tiene un mismo tratamiento, no es lo mismo la luz que sale de una bombilla que la que recibimos del sol o la que genera una linterna o flash del móvil. De ahí que vamos a distinguir tres tipos de luces que trataremos de diferentes formas [23]:

- **Luz direccional**: En este tipo de luz tenemos una fuente de emisión en forma de infinitos rayos paralelos apuntando hacia nuestros objetos, por lo que tenemos que tener en cuenta la dirección de éstos, para poder calcular su difusión. El resto de parámetros que debemos configurarle son la intensidad de las tres componentes vistas anteriormente, que se aplicarán a éstas para sus respectivos cálculos explicados anteriormente. Este tipo de luz es el más sencillo de entender ya que solo necesitamos conocer la dirección de los rayos.
- **Luz multidireccional**: Ahora ya tenemos un punto fijo de donde se van a generar los rayos de luz, por lo que ahora no tenemos una dirección si no infinitas. Por lo que ahora no tendremos un parámetro con la dirección, nos toca introducir un parámetro que indique la posición de donde se va a emitir la luz, sumado a la intensidad de sus tres componentes. Ahora tenemos que añadir un parámetro más, la atenuación de la luz, que se encargará de determinar la intensidad de nuestra luz en función de la distancia a la que está del foco emisor.

Antes de hablar de los siguientes parámetros tenemos que hablar de una cierta propiedad que tiene la luz, la atenuación [33]. Que es la pérdida de intensidad que sufre ésta al ser transmitida por un medio, ya pueda ser agua, aire o algún sólido. En este caso nos basaremos en las fórmulas aplicadas a la atenuación en el aire ya que no vamos a renderizar agua en nuestro programa. Tenemos pues una función, inversamente proporcional a un polinomio cuadrático, que define esta propiedad. Aquí tenemos la fórmula:

$$A = \frac{1}{(K + L * distancia + Q * distancia^2)}$$

Donde K, L y Q son los términos constante, lineal y cuadrático del polinomio respectivamente.
Tendremos que aplicar el resultado de esta fórmula para poder aplicarle a cada fragmento la intensidad de luz que le está llegando en estos momentos. Por lo que tendremos este parámetro dado por tres constantes, una para la parte constante, otra para la lineal y la otra para la cuadrática. Una vez cargados estos parámetros tendremos la luz multidireccional lista para renderizar.

- **Luz de Spot**: Esta luz es la más complicada de las tres, ahora el foco de emisión es un punto con una dirección y un ángulo de emisión que determinarán un cono donde se iluminarán los objetos que acaben dentro de éste, en nuestro caso el punto y la dirección serán los de la propia cámara, que tendrá que cargarle su posición y dirección a los parámetros de posición y dirección de esta luz. También tenemos que determinar el ángulo de emisión, lo cual nos obliga a tener que contar con un parámetro que indique este ángulo, es decir ϕ o mejor conocido como ángulo de corte, que nos indicará el ángulo que forma la dirección frontal de la cámara y el vector que une cualquier punto de la circunferencia de la base del cono con nuestro punto emisor.

Una vez conocidos todos estos parámetros el mecanismo es bastante sencillo, cogemos la dirección de la luz, que en este caso es la diferencia entre el punto del fragmento del objeto y el punto de emisión, y vemos el ángulo que forma con la dirección frontal, si este ángulo se encuentra entre $-\phi$ y ϕ tendremos que el objeto podrá iluminarse y cargar los valores de sus componentes (Difusión,Ambiente y Especular, que también como en los otros vendrán como parámetros) e iluminar el objeto. Si ese ángulo no está en ese rango el objeto no recibirá esta luz.

Ahora si queremos que la diferencia entre lo que está iluminado y lo que no lo está, debemos de saber que existen varios métodos para suavizar esta diferencia. Nosotros vamos a utilizar el siguiente, tenemos un cono exterior engendrado por ϕ que es el que hemos utilizado anteriormente, y otro cono más pequeño con un ángulo de corte γ, como siempre si el ángulo que forman la diferencia del foco con el punto y la dirección

frontal se sale del intervalo entonces se deja de iluminar, pero si está dentro se aplica la siguiente fórmula que calculará la atenuación:

$$I = \frac{\theta - \gamma}{\phi - \gamma} \tag{1}$$

Donde θ es el ángulo entre la diferencia del punto con el foco, e I es el valor de atenuación resultante que se multiplicará por las componentes de la luz para generar el resultado final.

3.3.2. Modelo de Bling-Phong

Ahora que ya tenemos el modelo de Phong explicado, sabemos todo su funcionamiento y cómo implementarlo, toca hablar de un cierto problema que éste genera y que nos obliga a buscar otras alternativas si queremos mejorar la iluminación de nuestra escena [24]. Si nos encontramos con un caso en el que el observador esté orientado con la figura de tal manera que forme con su normal un ángulo de más de 90 grados, entonces la componente especular va a tener un ligero problema, al ser el producto escalar negativo la potencia va a darnos algo completamente nulo y no vamos a recibir esa componente especular. Ésto para las demás componentes no es ningún problema, pero para ésta perdemos gran parte de la información del objeto que estamos observando.

Es aquí cuando introducimos el modelo de Bling-Phong que va a solventar este problema de una manera muy ingeniosa: Si cogemos en la componente especular el vector de la dirección de la luz y el de la vista, calculamos el vector que los separa equitativamente, es decir, el vector mediatriz. Entonces tenemos que si este vector se acerca mucho a la normal, percibiremos el reflejo del rayo mucho mejor que si está más alejado, y éstos no van a distar nunca más de 90 grados (Ya que eso solo pasa si fuera traslucido o transparente, caso que no tratamos) por lo que el producto escalar siempre va a dar positivo y es el que vamos a utilizar para elevarlo a la potencia del brillo del material. El resto de componentes se calculan exactamente de la misma manera que en el otro modelo, luego ya tenemos todo el conocimiento básico sobre los dos modelos de iluminación que vamos a utilizar.

Figura 11: Diferencia entre el modelo de Phong (Derecha) y el de Blinn (Izquierda).

3.4. Shaders de sombreado

Ahora entramos de lleno en efectos lumínicos más complicados, en este caso nos toca abordar las sombras y por tanto tenemos que hacer uso de los FrameBufferObjects, ya que en todos los ejemplos de sombras que vamos a utilizar, vamos a tener que usar un FrameBufferObjetcs para pintar una textura que nos indique dónde poner las sombras en nuestra escena. Antes de eso, hacemos un inciso rápido sobre qué consiste un FBO y para qué se pueden utilizar, ya que no va a ser la primera vez que hagamos uso de éstos para generar algún efecto lumínico.

Conocemos como FBO a la combinación de buffers destino encargados de recibir las imágenes renderizadas por nuestros shaders, como pueden ser buffers de profundidad o de color que son los que nos interesan. El FrameBufer numero 0 es el que está siempre por defecto y es en el que siempre se renderizará la escena que va a salir en pantalla, el resto son los que generemos a lo largo de nuestra ejecución, en ellos se puede almacenar una gran cantidad de información, ya sea el color de la escena que le estemos cargando o la misma profundidad de ésta. Para crear un FBO necesitaremos de una función conocida como glGenFramebuffers y para vincularlo al renderizado actual tendremos que utilizar glBindFramebuffer con el número de FBO que queramos utilizar en ese momento. Una vez creado podremos vincularle una textura vacía para que la dibuje, ya sea como una textura de profundidad (Usando el identificador GL_DEPTHATTACHMENT) o como una de color (Usando GL_COLORATTACHMENT), al vincular dicha textura con el FBO [32]. Una vez configurado, solo tendremos que renderizar nuestra escena con éste vinculado, y pintará sobre nuestra textura la profundidad o el color de la escena que podremos utilizar para el renderizado final. En la siguiente imagen podremos ver como se configura un FBO:

Ahora que ya nos hemos introducido en la herramienta más importante a la hora de generar nuestras sombras, vamos a separarnos en dos casos muy distinguidos: Sombras generadas por una luz direccional y las que son generadas por una luz multidireccional:

3.4.1. Sombreado en una dirección

La idea es bastante ingeniosa, generar una textura de profundidad pero en vez de utilizar nuestra cámara como referencia como se hace siempre, vamos a utilizar la posición y dirección del emisor, así podremos determinar que objetos se van a ver afectados por la sombra y cuales se van a poder ver iluminados [26].

Un breve inciso para aclarar, en la práctica estas sombras se van a utilizar para las luces direccionales que no tienen un punto de emisión, por lo que muy inteligentemente vamos a generar dicho punto cogiendo el vector de dirección y multiplicarlo por un número negativo muy grande en valor absoluto, al tratarse de un buffer de profundidad con proyección ortogonal las sombras se adecuaran perfectamente a este tipo de luz.

Empezamos creando el FBO que se encargará de generar nuestra textura de Sombra, para ello tendremos que vincularle una textura 2D vacía con la siguiente configuración:

Renderizaremos la escena completa para que se pueda generar esta textura, como hemos dicho anteriormente, se trata de una textura de profundidad con referencia en el punto de la luz. Por lo que tendremos que hacer unas transformaciones diferentes en un shader diferente para poder llevar ésto a cabo. Es aquí cuando tenemos que utilizar un shader de vértices con una matriz de proyección y otra de vista diferentes para que encaren la escena a como si la estuviera viendo el punto de la luz y no nosotros. Como las sombras se van a proyectar ortogonalmente tendremos que la matriz de proyección es una aplicación de ortogonalización que filtra los puntos en un rango ortogonal en vez del perspectivo. Para la matriz de vista tendríamos que usar la aplicación de GLM LookAt, que genera una matriz de vista en función del espacio visual que queramos generar, en nuestro caso como punto de origen el de la luz. Una vez sacadas estas matrices solo tendremos que usarlas en nuestro shader y nos generará la textura de profundidad buscada.

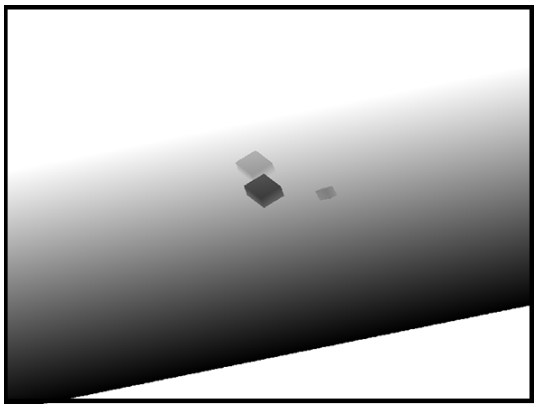


Figura 12: Textura del mapa Sombra renderizado.

Ahora que hemos renderizado nuestra escena en el FBO y generado nuestra textura de sombras, tendremos que cargarla en el shader definitivo que al hacer la última renderización generará la escena final con las sombras incluidas. Por lo que veamos que método se encarga de dibujar en el Fragment shader éstas, basándose en el ShadowMap que acabamos de generar. Antes en el Shader de vértices tendremos que calcular la posición del fragmento en el espacio con la posición de la luz como referencia, para poder consultarlo luego en la textura, una vez obtenido este fragmento transformado tendremos que convertirlo a NDC ya que el FBO ha convertido la textura a este formato, ahora ya si podemos consultarlo y ver la profundidad más visible en ese fragmento, si la profundidad del fragmento actual es menor estricto está bajo una sombra y no se iluminará, si es igual si se podrá iluminar.

Si no somos cuidadosos y no utilizamos una cierta cota de profundidad, no nos dibujará la sombra y nos aparecerá una especie de acné de sombras, por lo que deberemos ajustar un cierto parámetro de cota superior para que si la profundidad se encuentra por debajo no lo considere sombra. Calcular esta cota requiere de ciertas fórmulas que dan este valor en función de cómo le llegue el rayo de luz al fragmento.

Para suavizar la diferencia de la sombra con las superficies iluminadas utilizaremos el algoritmo PCF, que tratará de dibujar ciertas porciones de fragmento, que vienen en función de lo que permita la textura de sombras, llamados texels. Si en ese fragmento sus texels contenidos algunos se meten en una sombra y otros no, se calculará la media aritmética de los que se meten y éstos determinaran el grado de sombra de 0.0 a 1.0 que tendrá el fragmento. Con esto los fragmentos que se sitúen en los limites se pintarán en tonos grisáceos que suavizarán notablemente el contraste.

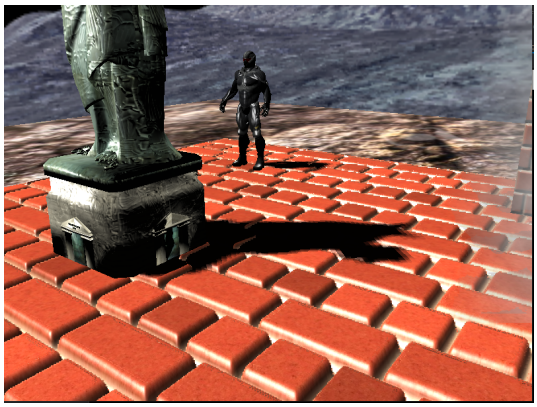


Figura 13: Renderizado final con la sombra dibujada.

Ahora ya hemos visto cómo generar el primer tipo de sombra, el siguiente tipo tendrá un procedimiento bastante similar pero en vez de utilizar una textura 2D, utilizará una cúbica

que devolverá ciertos parámetros en función de la posición que tenga el fragmento en la escena original.

3.4.2. Sombreado multidireccional, usando el Mapeado Cúbico.

Ahora en vez de tener una luz unidireccional disponemos de un punto de emisión con infinitas direcciones hacia todos los lados, es decir un tipo de luz puntual, ahora no solo tenemos que ver la profundidad en un única textura ya que no valdría para cubrir todos los casos, tenemos dos opciones, hacer 6 veces el procedimiento anterior de forma que proyectemos con una aplicación perspectiva de 90 grados en cada dirección del punto de emisión lo cual nos obliga a renderizar la escena previamente seis veces, que es completamente contraproducente en muchas tarjetas gráficas. O bien, coger un nuevo tipo de textura que haga este procedimiento de una tirada y solo necesitemos renderizar la escena una única vez. Es aquí donde las texturas cúbicas entran en acción [27].

Ahora tenemos que cambiar la manera de configurar el FBO para que dibuje sobre esta textura cúbica, por suerte openGL puede configurar texturas cúbicas a partir de 6 texturas 2D utilizando GL_TEXTURE_CUBE_MAP_POSITIVE_X + i como target, éste vinculará cada textura en su correspondiente cara. El resto de la configuración es similar que lo anterior.

Como hemos mencionado anteriormente, tenemos que hacer el renderizado una sola vez por lo que el shader que lo gestione debe ser distinto al anterior, las diferencias con el anterior van a ir respecto al Shader de fragmentos y al uso de un Shader Geométrico entre medias para proyectar en cada cara del cubo la posición del fragmento. En el shader de fragmentos vamos a hacer uso de la variable glFragDeph, que se encarga de almacenar en el FBO la profundidad del fragmento que se está renderizando. Ahí le asignaremos el valor de profundidad que le corresponde si el punto de donde parte la luz le estuviera mirando sobre una de las caras.

Esto último mencionado nos abre una duda, si el glFragDeph solo guarda la profundidad mirando hacia una cara, ¿Qué pasa con el resto? Es aquí donde entra a jugar el shader geométrico que gestiona el paso previo, este se encargará de proyectar el fragmento modelado sobre cada una de las caras y enviarle cada una de estas seis proyecciones al Shader de Fragmento por separado para que dibuje en cada cara la profundidad indicada. Con esto se pintarán todas las caras y se generará una textura cúbica lista para poder utilizarse.

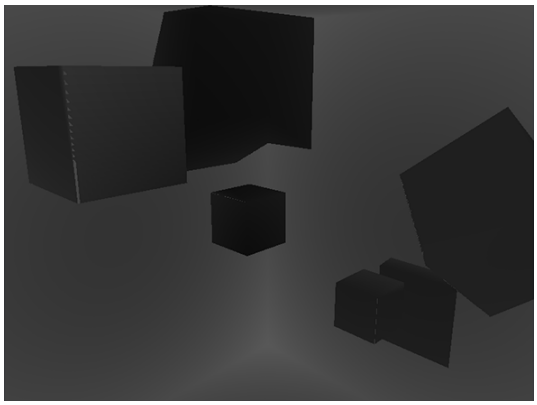


Figura 14: Textura del cubo Sombra renderizado.

También cabe resaltar que la matriz de proyección se generará de una manera diferente, en vez de usar una proyección ortogonal usaremos una perspectiva de 90 grados y una matriz de vista que mire a cada una de las caras, lo cual resulta en un Array de 6 matrices que el ShaderGeométrico pasará al fragmento la posición que le corresponda a su respectiva cara.

Ahora solo nos queda renderizar la escena de la misma manera que si utilizáramos una sola textura 2D pero con un ligero cambio, ahora no hace falta convertir el fragmento al espacio de visión de la luz, ya que la textura cúbica de la sombra almacena la profundidad de cada fragmento en toda la escena, independientemente de la cámara que le esté mirando, luego solo deberemos meter la posición únicamente modelada y la textura nos indicará que profundidad tiene. El resto de los pasos son completamente similares que en el caso anterior, salvo el PCF que ahora al tener una textura cúbica tendrá que tener un procedimiento algo más eficiente.

Con esto hemos terminado la parte del sombreado de la escena, ahora nos adentramos en temas de profundidad, es decir, con la luz determinar la profundidad de una superficie y hacerla parecer algo más real. En este caso tenemos dos tipos de mapeado para lograr este efecto que explicaremos en la siguiente parte.

3.5. Profundidad en una textura

Hasta ahora hemos utilizado texturas completamente planas, que si las vieras de lado notarias una sensación de completa irrealidad, esto a la hora de diseñar un escenario de cualquier juego daría lugar a una sensación confusa del entorno donde estamos jugando, es por eso que tenemos que mapear nuestras texturas de una manera diferente para que puedan hacernos percibir que los objetos que tienen una textura, tienen una cierta profundidad que no los hace 100 % planos. Esto se logra utilizando técnicas de mapeado que se encargan de proyectarnos la luz que deberíamos ver si percibimos cierta profundidad en una superficie, en nuestro caso vamos a ver dos: Mapeado con Normales y Mapeado por paralaje:

3.5.1. Mapeado de normales

El principal motivo por el que percibimos que un objeto está completamente plano es porque en toda la superficie, la normal es siempre la misma, y si pasáramos esto a la realidad nunca es cierto, toda superficie va variando su vector normal como por ejemplo un plano rocoso que dependiendo de como sobresalten las piedras tendrá una normal mirando a un sitio u otro. Por lo que tenemos que, de alguna manera, poder guardar en una superficie estas variaciones que vayan apareciendo en la textura, la primera idea que pensaríamos sería pasarle por memoria las distintas normales por el buffer, cosa completamente contraproducente, pero si pensamos un poco en qué utilizamos para determinar una sombra, se nos ocurre que una textura podría darnos la solución que estamos buscando y efectivamente así es.

Ahora tenemos que encontrar la manera de representar las normales en dicha textura, podemos utilizar los colores rgb a modo de cooordenadas. El azul representará el eje z, el verde al eje y y el rojo al x. Ya una vez tenemos esta representación tendriamos ya la solución perfecta para pasar nuestras normales, solo tendríamos que consultar las coordenadas de textura del fragmento en la textura normal y de ahí obtener la normal correspondiente para calcular la iluminación del objeto con los modelos de iluminación explicados anteriormente. Con esto tendríamos las profundidades de nuestra textura bien definidas y le dará una sensación de realismo mucho más eficaz [28].

Pero como todo lo que en principio suena sencillo tiene un ligero problema, que la superficie no siempre va a estar modelada de tal manera que la textura de las normales represente correctamente con los colores rgb la dirección de los ejes x,y,z, por lo que nos obliga a tener que hacer una transformación previa de las posiciones para que los colores de la textura puedan corresponderse con el sentido de los ejes.

Si consiguieramos llevarnos las coordenadas de nuestro fragmento, la posición de la luz y la de la camara a este espacio podremos corresponder la normal con los colores rgb de la textura de normales. Tenemos que elegir bien la transformación y lo que queremos, necesitamos un espacio en el que la normal de nuestro fragmento siempre esté mirando hacia eje Z , la tangente hacia el eje Y y la binormal mirando hacia el X. Si generamos un espacio en el cual éstos sean la referencia conseguiríamos el objetivo buscado, por lo que la solución está en ver la posición en el espacio global de los vectores normal, tangente y binormal, una vez los tengamos, y finalmente crear con estos resultados la matriz de cambio de coordenadas que mueva el eje Z al eje de la Normal, y así respectivamente con los demás, a este espacio se le conoce como Espacio Tangente a una superficie. Si la tangente y la binormal nos vienen en el array de vertices no hay problema, pero si no vienen podemos calcularlos utilizando ciertos metodos bastante sencillos de comprender [36].

Una vez que conseguimos nuestra matriz de cambio solo tenemos que pasar por ella los tres puntos antes mencionados y pasárselos al Shader de fragmentos para que con ellos y la posición en la textura normal que le corresponde al fragmento se realicen los cálculos ya conocidos para obtener la iluminación adecuada de éste. Con esto damos por terminada la primera forma que tenemos de mapear la profundidad de las texturas, la siguiente es algo más complicada.

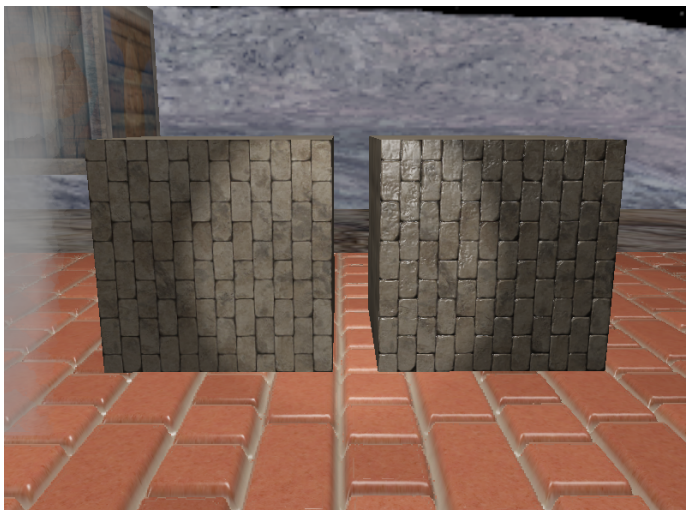


Figura 15: Diferencia de una textura con normales a una sin normales.

3.5.2. Mapeado por paralaje

Ahora si en vez de centrarnos en las normales utilizasemos otra manera de ver cómo de inclinada o profunda está una textura en nuestra superficie y usaramos otro tipo de textura para indicar esto, como por ejemplo la profundidad de un fragmento en las coordenadas de ésta, es decir usar una textura de profundidad. Es aquí cuando introducimos el Mapeo por paralaje, que consiste en ir viendo las profundidades de un fragmento a través de una textura que indique éstas, pero hay un inconveniente. ¿ Cómo conseguimos que la luz nos muestre la parte de la textura que estamos observando correctamente solo sabiendo lo profunda que está? La respuesta está en coger la dirección formada por el punto de vista y el fragmento, multiplicarla por un parámetro en función de la profundidad y desplazar las coordenadas de textura justo ese resultado para así lograr hacer creer al observador que está viendo la parte correcta de la textura [29].

Ahora que ya sabemos el funcionamiento de este procedimiento vamos a detallar ciertos problemas, el primero que debemos tener un parámetro que reduzca el producto por la profundidad, porque se nos puede alejar demasiado a donde debería acercarse. Si tenemos en cuenta relieves poco inclinados el mapeo por normales funciona perfecto y realiza los desplazamientos de coordenadas correctamente pero si la superficie está muy inclinada nos va

a generar muchísimos problemas que van a hacer que nuestra figura parezca de todo menos real. Por otra parte, también cabe resaltar que este tipo de mapeo tiene una ventaja frente al otro, nos hace ver el objeto más real ya que la textura al girar con nosotros nos hace ver la profundidad del objeto en función del lugar donde lo veamos, lo cual hace que si nos movemos por distintas partes de la superficie veamos que hay partes que se pueden ver desde un lado y desde el otro no. Lo podemos apreciar en la siguiente imagen tomada de nuestro pequeño motor gráfico implementado durante el proyecto:

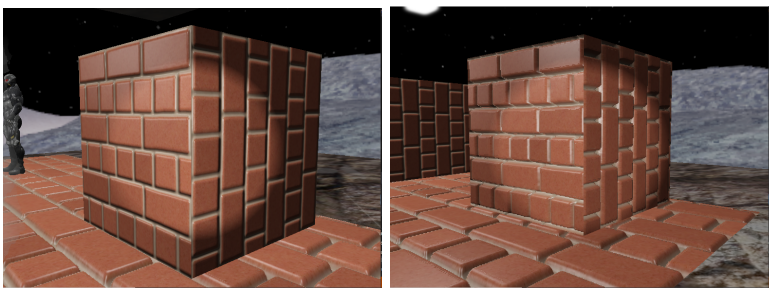


Figura 16: Diferencia del mapeado por normales (Izquierda) con el mapeado por paralaje (Derecha).

Para solventar el problema de los relieves inclinados existen dos algoritmos, uno es la mejora del otro así que explicaremos ambos en el orden correspondiente. El primero es el paralaje por pasos, que consiste en coger la dirección de vista y en vez de chocarla contra la superficie solamente, y sacar de ahí sus coordenadas para sacar la profundidad como hemos hecho siempre, ir bajando niveles y ver la profundidad en esos niveles de los puntos de corte que tengan con el vector de vista, iremos bajando de profundidad hasta que lleguemos a un punto en el que la profundidad en ese punto sea superior que el mismo punto y por tanto nos hayamos pasado del punto buscado.

Este algoritmo mejora bastante las pendientes elevadas en nuestras superficies pero se sigue quedando corto, es aquí donde tenemos el siguiente algoritmo, su mejora. El algoritmo de paralaje por pasos con interpolación, consiste en hacer los mismos pasos que el anterior, pero ahora una vez encontremos un punto que su profundidad sea mayor cogeremos este y su anterior en la sucesión, calcularemos la recta que cruza estos dos puntos. Una vez tengamos la recta, el corte que tenga ésta con el vector de vista será el punto que estamos buscando y el que más cerca se encuentre de la profundidad exacta. Usaremos este punto como coordenadas de textura y así obtener el mapeado por paralaje deseado.

Ahora ya estamos listos para poder generar texturas con profundidades y adaptadas a la realidad, el mapeado por paralaje como bien dijimos es muy difícil de hacer con exactitud en superficies muy puntiagudas, por lo que se suele utilizar para mapear paredes, tejados y suelos, es decir, planos que se comportan bien ante este tipo de mapeados. Sin embargo el

mapeado normal si que se utiliza bastante a la hora de sacar profundidades de los modelos 3D realizados por artistas, es más, las mayas que vamos a cargar en nuestros modelos con Assimp van a utilizar este tipo de mapeado.

3.6. Floración de la luz

Ahora en vez de centrarnos en renderizar solamente los objetos de la escena en un FBO, vamos a encargarnos de renderizar las componentes más brillantes de la escena también, o sea aquellas componentes cuyo brillo excede de 1.0 en el formato rgb de una de sus componentes y no son bien representadas. El ejemplo más básico y que a todos se nos viene a la cabeza es la luz de un foco, ese destello que emite que parece que se ve borroso, nuestro objetivo va a ser renderizar correctamente esa parte de la escena. Para ello tenemos la técnica de la Floración de la Luz (Light Blooming) que se basará en usar técnicas basadas en el HDR(Rango de color dinámico) para poder captar las componentes con brillos excesivos y ajustarlas a la escena original.

Para ello como bien hemos mencionado, tenemos que conocer en qué consiste la técnica del HDR, y que usos tiene a la hora de generar gráficos de alta luminosidad. También deberemos conocer el uso del FBO, que ya conocemos pero esta vez con un RenderBufferObject que se encargue de almacenar la profundidad, ya que tendremos que vincular texturas de color. Explicaremos este tipo de configuración de nuestro FBO más adelante. Ahora vamos a centrarnos en los procedimientos HDR para poder entender mejor todo el proceso.

3.6.1. HDR o Rango de color dinámico

Las tecnicas de alto rango dinamico son aquellas que se utilizan cuando los shaders generan colores que superan el valor de 1.0 y no son capaz de renderizarlos, dejando solamente un blanco que no se distingue de un blanco cualquiera, este problema nos surge en zonas de alta luminosidad como puede ser cuando nos situamos cerca de un foco o cuando varios focos apuntan a un mismo sitio. Estas técnicas van a tratar de poder detectar estas zonas y poder renderizarlas de tal manera que se puedan distinguir en la escena y dar una mejor visualización de ésta al usuario [30].

Tenemos entonces que tener en cuenta unos ciertos detalles para poder realizar estas técnicas correctamente, una es el uso de los FBO para poder renderizar una escena por separado con todos los brillos localizados y la otra es que éstos puedan soportar la alta gama de brillos de estas partes para poder almacenarla en el buffer. Para ello utilizaremos un colorBuffer con formato RGB en coma flotante, es decir, cuando generemos el colorBuffer tendremos que usar el tag GL_RGBA32/16F para especificarle que soporte este tipo de luces. Ahora solo tendremos que vincularlo y ya tendremos un FBO listo para renderizar las zonas de la escena que más brillos contengan.

Cabe añadir un inciso para explicar qué es la corrección gamma [25]. Un problema que actualmente tienen muchas pantallas a la hora de emitir el brillo, que al adaptarlo para los ojos humanos se pierde una proporción no lineal de brillo que afecta a los gráficos que estamos creando, en otras palabras, estamos proporcionando una cantidad de brillo inferior a la que deberíamos. En las pantallas normalmente esta adaptación se suele adaptar con una potencia a la cual denominaremos γ , es decir que el brillo de los colores renderizados saldrá elevado a dicha potencia por pantalla oscureciendo así el color reproducido por pantalla. La corrección gamma consistirá en coger nuestro resultado y aplicarle la inversa de esa potencia, es decir, elevarlo a $1/\gamma$ para que el valor se autocorrija al salir por la pantalla y salga su brillo correctamente. Existen dos maneras de aplicar la corrección gamma: Una activando el buffer GL_FRAMEBUFFERS_RGB, y la otra aplicando esta potencia directamente en el shader de fragmentos. El valor de γ depende de la pantalla que estemos utilizando, en la pantalla de mi ordenador funciona aproximadamente con 1.4, pero el valor suele ser casi siempre 2.2 en la gran mayoría de pantallas.

Ahora que tenemos nuestra textura con la escena pintada en ella solo tendremos que renderizar la escena original corrigiendo los excesos de ésta, debemos recordar el uso de la corrección gamma siempre que podamos para compensar el bajón de brillo, para la corrección de brillos usaremos la siguiente formula que acercará a 1.0 los valores más luminosos y acercará a 0.0 los más oscuros:

$$Color = 1{,}0 - e^{-E*HDR_{valortextura}} \tag{2}$$

Donde $HDR_{valortextura}$ es el valor que tomamos de la textura renderizada anteriormente, E es un valor de exposición que determinaremos según como queramos distinguir la escena, si es bajo los brillos se distinguirán mejor y los oscuros peor. Esto nos dará una textura que distinguirá los brillos y podremos distinguir las zonas de alta luminosidad de tal manera que no parezca que estamos en la Antártida. Ahora que ya conocemos esta técnica estamos listos para ver como tratar la floración de la luz [31].

3.6.2. Floración

Tenemos que generar los destellos que producen las zonas luminosas de nuestra escena, por lo que tendremos que basarnos en las técnicas antes descritas y generar un FBO que renderice sobre una textura las partes más iluminadas que superen la cantidad de 1.0. Pero también necesitamos la escena original por lo que vamos a asociar a nuestro FBO dos attachments o canales de salida asociadas a dos colorBuffers, una que se encargue de escribir en una textura la escena tal y como debería escribirse en la pantalla y la otra que se encargue de escribir las partes más iluminadas de esta, además añadiremos un RBO(RenderBufferObject) a nuestro FBO con un dephBuffer para no tener problemas dibujando una textura que no cumpla las profundidades y nos dé lugar a una textura errónea.

Ahora que tenemos dos texturas diferenciadas debemos interesarnos en la de las luces, ésta tendremos que procesarla con dos FBOs, que se encarguen de difuminarla para que así

consigamos un efecto de destello que es lo que estamos buscando, hay que tener cuidado cuando generemos esta nueva textura y aplicarle antes las técnicas de HDR para que se puedan distinguir bien estos destellos. Ahora vemos como construimos este nuevo FBO doble, que va a consistir en otros dos colorBuffer asociados cada uno a un FBO, éstos solo van a tener un canal de salida vinculado, estos colorBuffer se van a ir encargando de almacenar de uno en otro las distintas veces que tengamos que renderizar la escena utilizando el algoritmo que vamos a explicar a continuación, además de que la primera iteracción parte de la textura de luces dibujada por el FBO del principio. La textura dibujada resultante veremos más adelante como la mezclamos con la original, que está pintada en la otra textura del primer renderizado.

El algoritmo que utilizamos para generar la difuminación es el conocido desenfoque Gaussiano, que se basa en los valores de una campana de Gauss para calcular los grados de difuminación de un fragmento. Funciona mediante iteraciones de renderizado, es decir a cada iteración un renderizado de esta textura. Caben resaltar unos detalles, el primero que obviamente estos FBOs los utilizamos para realizar estas iteraciones, lo que escriba un ColorBuffer se lo pasa a la textura que se usará para pintar el otro en la siguiente iteración y así hasta que se termine, entonces se tendrá que indicar cual de los dos tiene la textura resultante de la última iteración. Lo segundo es que necesitamos un shader especial para realizar este algoritmo de desenfoque que renderice un rectángulo del tamaño de la pantalla para pegar la textura resultante en éste, siempre se tiene que renderizar algo por lo que elegimos un rectángulo que ocupe toda la pantalla y en el que dibujemos el resultado final, para el último paso haremos lo mismo.

Ahora podemos explicar este algoritmo que se realizará en el shader de fragmentación, primero como hicimos en el PCF dividir toda la pantalla en texels para poder medir la distribución de manera correcta, también debemos de tener el array de pesos gaussianos que salen de la gráfica de la distribución normal. Ahora tenemos que calcular la posición del fragmento actual, su valor en la textura y finalmente multiplicarlo por el peso gausiano más grande, una vez hecho esto tenemos que movernos entre los ejes X e Y por cada texel, cada vez que avancemos sobre un eje tendremos que calcular el valor de esa nueva posición en la textura y multiplicarlo por el siguiente valor del array gaussiano. En nuestro caso avanzaremos 5 veces hacia la derecha, cinco hacia la izquierda, cinco hacia arriba y cinco hacia abajo, cargando todos los resultados en un sumatorio y devolviéndolo como resultado final. Una vez hecho esto el colorBuffer lo pintará en una textura que pasará como parámetro para que el otro buffer escriba otra textura a partir de esta en la siguiente renderización. Una vez renderizado unas 15 veces, se habrá conseguido difuminar lo suficiente como para poder avanzar al siguiente paso.

Ahora tenemos por un lado la textura con la escena original pintada y por el otro el resultado de la textura de las partes iluminadas difuminada, es hora de mezclarlas, esto se debe de hacer sumándolas fragmento a fragmento. Ahora tenemos que tener cuidado y

utilizar las técnicas HDR para poder distinguir la parte del brillo en la textura resultante final, además de que es aconsejable una corrección gamma para que el brillo se acople a la pantalla, una vez hecho esto tendremos nuestra escena final resultante con los destellos de las luces mucho más vistosos y agradables a la vista. Un inciso, para hacer esta mezcla de texturas tenemos que utilizar otro shader nuevo que dibuje sobre un quad que ocupe toda la pantalla la escena final resultante.

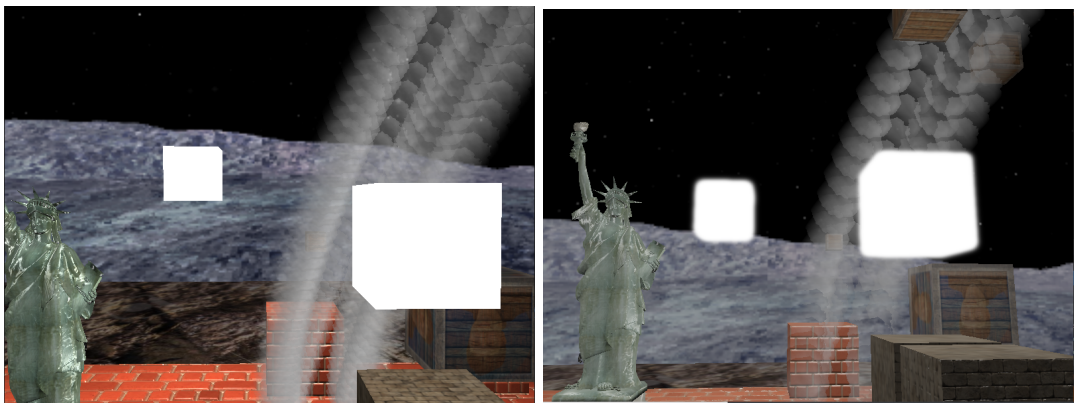


Figura 17: Diferencia de una escena sin floración (Izquierda) con una que si la tiene (Derecha).

Una vez visto el efecto más completo de todos, vamos a introducir al último que vamos a usar en el proyecto, el encargado de generar todo tipo de efectos que involucren gases, partículas, destellos, etc. Introducimos así al shader encargado de renderizar los sistemas de partículas.

3.7. Sistemas de partículas

En esta última parte vamos a ver como renderizar un sistema de particulas basandonos en un modelo que utiliza las shaders geométricos para generar un plano, o mejor dicho un cuadrilátero, que gira junto a nosotros para darle a estas particulas el efecto de estar en 3D. Existen varios metodos para implementar un sistema de particulas, uno puede ser cargando todas las partículas en el buffer de vertices para que se rendericen o instanciando todo el cuadrilátero, parámetros, posiciones y cualquier atributo que se requiera mucho antes de pasar por los shaders para que se renderice sin necesidad de pasar por un shader geométrico [35]. Pero nosotros queremos utilizar el shader geométrico debido a que nuestra tarjeta puede soportar su peso y queda un resultado bastante interesante.

Nuestro método va a calcular tanto el grado de transparencia, como la posición de la partícula en el mismo proceso de renderizado dentro del shader. Por lo que lo único que tenemos que hacer es cargar en el buffer de entrada tres parámetros, la posición inicial, la velocidad inicial y el tiempo de vida, además de tener una clase que gestione el tiempo en el

que nace la partícula para que se calcule cuando debe desaparecer, ésto lo haremos mediante un valor uniform. Del resto ya solo se tiene que preocupar el shader de realizarlo.

Para programar el shader vamos a basarnos en un modelo encontrado en una página [34] el cual vamos a versionar para que sea compatible con nuestro motor y cómo gestiona éste las partículas. Como bien hemos dicho anteriormente, el shader de vértices recibe del VAO la posición inicial, velocidad y tiempo de vida, además de tener como uniform la aceleración de la partícula, el tiempo de nacimiento de la partícula y el tiempo actual. El shader va a calcular con éstos tiempos el actual y lo va a usar con una fórmula parabólica con aceleración. Una vez calcule la nueva posición calculará si el tiempo ha superado el tiempo de vida para que en las siguientes etapas se descarte la partícula.

Cabe destacar que utilizaremos alpha blending para que las partículas más viejas que lleguen al shader de fragmentos se pinten mucho más transparentes que a las jóvenes, por lo que tendremos que activar las funciones de alpha blending y usar la función GLONEMINUSSRCALPHA para los cálculos del blending. El shader de fragmentos se va a encargar de dibujar la imagen que le hayamos cargado para generar las partículas y le aplicará el blending según corresponda.

Por último entre medias, el shader geométrico es el que se va a encargar de hacer lo más importante, generar el cuadrilátero que se va a encargar de ponernos las partículas mirando hacia nuestra cámara. Para ello tenemos que coger la posición de la partícula y pasarla por la matriz de modelado y vista que vendrán en función de la posición de nuestra cámara y de donde queramos tener ubicado nuestro foco de partículas. Una vez obtengamos el punto del espacio visual tendremos que calcular el cuadrado que lo rodea y generar una primitiva TriangleStrip que se mandará al shader de fragmentos, aunque los puntos de esta primitiva tendremos que pasarlos antes por la matriz de proyección, para que se pueda ubicar en nuestra pantalla 2D. La posición del fragmento se pasará también junto a su alpha correspondiente para que sepa cuanta transparencia se le tiene que asignar.

Ahora solo tenemos que indicar el número de partículas que queremos renderizar y en cada iteración del bucle de renderizado soltaremos una partícula nueva hasta que salgan todas. En cada iteración se renderizarán las que ya están lanzadas con sus nuevos tiempos actuales para calcular su nueva posición o su desaparición. Con esto hemos terminado todo lo que necesitamos saber sobre shaders para poder generar los efectos de los que va a disponer nuestro motor gráfico.

Figura 18: Renderizado del efecto cortina de humo usando un sistema de partículas.

4. Implementación del motor gráfico

En esta sección vamos a tratar de exponer el resultado de toda la investigación que ha llevado el proyecto, poder generar mi primer motor gráfico capaz de renderizar una escena, cargarle modelos exportados de un cargador, en este caso Assimp, que gestione la iluminación siguiendo dos modelos diferentes, pueda diferenciar los focos de emisión de luz, disponga de texturas que puedan aportar profundidad a un objeto y respondan a la iluminación, además de otros efectos como las sombras y los sistemas de partículas. Tendremos un motor que podrá cargar una escena desde un archivo .txt de configuración, lo que nos dará la libertad de ponerle a nuestro antojo los objetos, modelos, parámetros lumínicos, las sombras y las partículas. Todos éstos podremos ubicarlos donde queramos que aparezcan en nuestra escena.

En este programa hemos decidido utilizar la programación orientada a objetos para poder gestionar de manera estructurada todas las componentes de la escena, y por tanto está estructurado por clases que representan cada componente de la escena, además de que la propia escena es una clase que tiene a todas estas componentes como atributo y se encarga de organizar su renderización, digamos que la clase escena va a hacer la función de controlador. Vamos a empezar hablando de todas las clases del programa y qué aportan sus atributos en estas.

4.1. Estructura de las clases que forman el motor

Para empezar vamos a tener un Main.cpp encargado de abrir la ventana de openGL y generar/abrir el objeto GraphicProgram encargado de ejecutar el programa donde se renderice nuestra escena, por lo tanto tenemos aquí dos objetos de vital importancia:

1. **GraphicProgram**: Esta clase va a ser la encargada de organizar todas las partes que se desarrollan en el escenario, desde su inicialización con la carga de todos los archivos

y moviendo sus datos al escenario hasta llevar el bucle de renderizado de la escena.

Tiene por atributos el objeto que representa a la ventana openGL, otro de clase Escena que se trata de la escena que se está renderizando y deltaTime/currentTime para controlar los Inputs del teclado. Tiene el método *processInput*(), encargado de recibir una entrada procedente del teclado y llamar al método de Escenario correspondiente a la acción que tenga asociada. También dispone de los métodos *cargaPrograma*() que solo transmitirá el tamaño de la ventana al escenario y *ejecutarPrograma*() que se encargará de cargar el Escenario con todos sus archivos de carga y entrar en el bucle de renderizado donde se renderizará continuadamente éste.

```
void ejecutarPrograma() {
    char * archivo_ver = "", *archivo_caja = "", *archivo_luces = "", *archivo_particulas = "", *archivo_modelo = "";
    archivo_ver = "figuras.txt";
    archivo_caja = "caja.txt";
    archivo_luces = "luces.txt";
    archivo_particulas = "particulas.txt";
    archivo_modelo = "modelo.txt";
    esc.cargarShaderAssimp("VshaderAss.vs","FshaderAss.fs", nullptr);
    esc.cargarShaderBloom("VshaderBloom.vs", "FshaderBloom.fs", nullptr);
    esc.cargarShaderBluring("VshaderBluring.vs", "FshaderBluring.fs", nullptr);
    esc.cargarShaderCajaFondo("VshaderCaja.vs", "FshaderCaja.fs", nullptr);
    esc.cargarShaderSombra("VshaderSombra.vs", "FshaderSombra.fs", nullptr);
    esc.cargarShaderSombraCubo("VshaderSombraCubo.vs", "FshaderSombraCubo.fs", "GshaderSombraCubo.gs");
    esc.cargarShaderParticulas("VshaderParticulas.vs", "FshaderParticulas.fs", "GshaderParticulas.gs");
    esc.cargarShaderFigura("VshaderFigura.vs", "FshaderFigura.fs", nullptr);
    esc.cargarShaderFocosLuz("VshaderFocosLuz.vs", "FshaderFocosLuz.fs", nullptr);
    esc.cargarEscenaCompleta(archivo_ver, archivo_caja, archivo_luces, archivo_particulas, archivo_modelo);
    while (!glfwWindowShouldClose(window))
    {

        float currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        processInput(window);

        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        esc.renderizar(glfwGetTime());

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
}
```

Figura 19: Método de ejecutarPrograma( ) en la clase GraphicProgram.

2. **Escenario**: Aquí tenemos la clase más importante de todo el motor, esta clase se va a encargar de renderizar todos los objetos gráficos en el orden correcto y a la vez cargarles todos los atributos necesarios para que éstos puedan hacerlo. Tenemos que destacar bastantes atributos que podemos enumerarlos en una amplia lista:

   a) **Nparticulas**: Número de partículas que se van a crear en la escena, por el momento solo van a tener un mismo foco de emisión todas por lo que solo necesitamos saber cuantas van a salir.

   b) **anchura**/ **altura**: Tamaños de ancho y alto de la ventana openGL.

   c) Enteros que indican que tipo de escena va a ser, en este caso tenemos: **Sombreado**, para indicar si vamos a tener sombras en nuestra escena, **TipoSombreado**

para indicar si tenemos sombras direccionales o tenemos sombras de luz puntual y por último, si queremos que haya floración en la escena con el atributo **blooming**.

d) Todos los shaders que se van a utilizar y que GraphicProgram se encargará de cargarlos llamando a sus métodos de carga, en este caso tenemos: **shaderFigura**, para los objetos que no son mayas cargadas de assimp, en este caso tenemos suelo, paredes y cubos, **shaderAssimp** para renderizar todas las mayas de Assimp, **shaderCajaFondo** para poder renderizar el fondo de la escena, **shaderFocosLuz** para poder poner en pantalla los distintos focos de luz puntual que queramos meter en la escena (5 como límite), **shaderSombra** encargado de renderizar las texturas de sombras, **shaderSombraCubo** para encargarse de renderizar las texturas cúbicas de sombras, **shaderBluring** para poder difuminar las texturas luminosas, **shaderBloom** para poder coger estas texturas y añadírselas a la escena original y por último **shaderParticulas** que es el encargado de renderizar en la pantalla nuestro sistema de particulas.

e) **iluminacion**: Atributo de clase tipo Luz encargado de gestionar toda la iluminación del escenario.

f) **shadow**: Atributo de tipo Sombra encargado de gestionar las sombras de la escena.

g) **objetos**: Lista de todas las figuras de tipo Entity, es decir las que no son modelos cargados con Assimp, que se han cargado en toda la escena.

h) **modelosAss**: Lista de todos los modelos cargados con Assimp que van a aparecer en la escena.

i) **fondo**: Atributo de tipo mapaCubo encargado de gestionar el fondo de la escena.

Una vez vistos los atributos toca meternos de lleno con los métodos que son la parte más importante de la clase, y por no decir de todo el programa ya que es donde se van a organizar todos los atributos para que acaben llamando correctamente a los shaders y renderizando toda la escena en el orden correcto, por lo que vamos a tener tres tipos de métodos, los métodos de carga que cargarán los datos a los atributos, los métodos encargados de gestionar los inputs y el método de renderizado que se encargará de hacer todas las llamadas a renderizar correctamente.

Los métodos de carga de la escena se pueden resumir en lo siguiente: GraphicProgram llama a los métodos de carga de los shaders cuyo funcionamiento consiste en cargar en estos la dirección donde se ubican sus respectivos shaders de vértices, fragmentos y geométricos si los tienen. Para cargar los datos de la escena tenemos el método *cargarEscenaCompleta*() que tiene las direcciones de los archivos de carga de cada parte de la escena, desde las figuras hasta las partículas que esta va a tener. Este método va a llamar a los distintos submétodos encargados de cargar diferentes archivos de manera independiente.

```
void cargarEscenaCompleta(char* vertices, char * caja, char * luces, char * particulas, char * modelo) {

    cargarVertices(vertices);

    cargarMapaCaja(caja);

    cargarLuces(luces);

    cargarParticulas(particulas);

    ifstream file;
    file.open(modelo);
    int numModelos;
    file >> numModelos;
    for (int k = 0; k < numModelos; k++) {
        char modeloAssimp[100];
        float x, y, z;
        file >> modeloAssimp;
        file >> x >> y >> z;
        glm::vec3 tr(x, y, z);
        file >> x >> y >> z;
        glm::vec3 esc(x, y, z);
        cargarModeloAssimp(modeloAssimp,tr, esc);
    }

    file.close();
}
```

Figura 20: Método para cargar los datos de la escena de un archivo.

El método de renderizado se va a encargar de ir llamando a cada atributo en su orden correspondiente para que vaya renderizando su correspondiente parte a la escena, primero se va a encargar de renderizar la iluminación de la escena, después las sombras si las hubiera, luego en caso de usar floración deberá de abrir el FBO correspondiente para que los objetos de la escena se rendericen ahí y cerrarlo después de que pase esto, más adelante renderizar los objetos y los modelos de Assimp con sus respectivos métodos en función de los datos de la escena. Una vez tengamos los objetos y modelos de la escena, renderizamos el fondo de pantalla y las partículas, por último si utilizamos floración deberemos de hacer la difuminación y el renderizado final de la escena para añadirle tal efecto.

```
void renderizar(float tiempo) {

    if (sombreado == 0) {
        iluminacion.renderizarLuz(shaderFigura, shaderAssimp, shaderFocosLuz, cam.Position, cam, 0, anchura, altura);
    }
    else if (tipoSombreado == 0) {
        iluminacion.renderizarLuz(shaderFigura, shaderAssimp, shaderFocosLuz, cam.Position, cam, 1, anchura, altura);
        shadow.renderizaSombra(shaderSombra,shaderFigura , shaderAssimp, cam, objetos, modelosAss, anchura, altura);
    }
    else {
        iluminacion.renderizarLuz(shaderFigura, shaderAssimp, shaderFocosLuz, cam.Position, cam, 2, anchura, altura);
        shadow.renderizaSombra(shaderSombraCubo, shaderFigura, shaderAssimp, cam, objetos, modelosAss, anchura, altura);
    }

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if (blooming != 0)
        iluminacion.abrirFBF();
    // Primero las figuras

    for (Entity obj : objetos) {
        if (sombreado != 0) {
            if (tipoSombreado == 0) {
                obj.cargarTexturaMapaSombra(shadow.getShadowMap());
                obj.renderizar(shaderFigura, cam.Position, cam, 1, anchura, altura);
            }
            else {
                obj.cargarTexturaCuboSombra(shadow.getShadowCubeMap());
                obj.renderizar(shaderFigura, cam.Position, cam, 2, anchura, altura);
            }
        }
        else {
            obj.renderizar(shaderFigura, cam.Position, cam, 0, anchura, altura);
        }
    }
```

```
        // Luego los modelos de Assimp
        for (Model mod : modelosAss) {
            // cargar la matriz modelo para ubicarlo.
            if (sombreado != 0) {
                if (tipoSombreado == 0) {
                    mod.Draw_mapaSombra(shaderAssimp, cam, shadow.getShadowMap(), anchura, altura);
                }
                else {
                    mod.Draw_cuboSombra(shaderAssimp, cam, shadow.getShadowCubeMap(), anchura, altura);
                }
            }
            else {
                mod.Draw_normal(shaderAssimp,cam, anchura, altura);
            }
        }

        iluminacion.renderizaFocos(shaderFocosLuz, cam, anchura, altura);
        fondo.renderizaCaja(shaderCajaFondo, cam, anchura, altura);

        if (Nparticulas != 0) {
            particulas.renderizarParticulas(shaderParticulas, cam, tiempo, anchura, altura);
        }


        if (blooming != 0) {
            iluminacion.cerrarFBF();
            iluminacion.renderizarBlur(shaderBluring,objetos.at(0)); // Con coger un entity cualquiera nos vale
            iluminacion.renderizarBloom(shaderBloom, objetos.at(0));
        }

    }
```

Figura 21: Método para renderizar una escena.

Ahora que ya hemos visto a fondo la clase Escenario nos vamos a los componentes que la forman, trataremos de explicarlos en orden de más importante a menos. Empezamos con los objetos que se van a renderizar en nuestra escena:

1. **GraphicObject**: Clase abstracta que servirá como modelo para todos los objetos que se rendericen en la escena, ya que estos deberán implementar de esta, para ello se caracterizará por tener un atributo *idObject* para poder identificar a cada objeto de alguna forma, además de tener dos métodos bastante diferenciados: $renderizar()$ y $cargarMatrizModelo()$, uno para poder renderizar el objeto en pantalla o en FBO, y el otro para generar una matriz de modelaje que ubique al objeto en la escena.

```
class GraphicObject {
public:

    virtual void renderizar(Shader shaderFig, glm::vec3 pos, Camera cam, int sombra, int w, int h) = 0;

    virtual void cargarMatrizModelo(glm::vec3 v1, glm::vec3 v2) = 0;

    int getIdObj() {
        return idObject;
    }

private:
    int idObject;

};
```

Figura 22: Clase abstracta GraphicObject.

Como hemos dicho anteriormente, todo objeto con vértices a excepción de las partículas, los focos y el fondo del escenario deberán heredar de esta clase para seguir unas mismas pautas, de los cuales al ser un motor pequeño tenemos dos que son los siguientes:

a) **Entity**: Esta clase va a abordar un gran peso en el proyecto porque es la encargada

de renderizar todas las figuras que le carguemos en el archivo de carga, cada figura va a estar asociada a una entidad de esta clase, por lo que ésta deberá gestionar los datos de esta figura para poder ponerla en pantalla correctamente. Por lo que tendremos que tener en cuenta el tipo de escena al que pertenece, el tipo de textura, si es una figura que se carga con tangentes y binormales, si es una figura que trata con sombras, si va a estar compuesta de algún material con características lumínicas distintas y cuantas texturas va a tener. De ahí que tengamos los siguientes atributos:

1) Todas las texturas que pueden ir asociadas con el objeto, texturas básicas, texturas especulares, normales, de profundidad o texturas de sombreado. Cada una tendrá un atributo reservado para tratarse.
2) **matrizModelo**: Atributo mat4 que se corresponde con la matriz modelo que va a modelar el objeto en la escena.
3) **VAO** y **qVAO** con sus respectivos buffers de vértices asociados, uno sirve para renderizar cubos y el otro superficies planas.
4) **tipoFigura**: Entero que sirve para indicar si la figura es un cubo o un plano.
5) **tipoTextura**: Entero que sirve para indicar que tipo de textura va a rellenar el objeto, con materiales o sin materiales.
6) **heightscale**: Parámetro que sirve para controlar el paralaje del objeto, ya lo explicamos en la sección anterior.
7) **brillo**: Parámetro de brillo que va a tener el material de nuestra figura si lo tiene.
8) **paralaje** y **TBN**: Enteros que sirven para indicar si la figura tiene mapeo por paralaje o por normales y la segunda es si acepta tangentes/binormales.
9) **numTexturas**: Número de texturas que se van a mezclar para pintarse en el objeto.

Ahora que comprendemos todos los atributos que forman parte de nuestra clase Entity, vamos a lo más importante, sus métodos, de los cuales hablamos rápidamente de sus métodos de carga para todos estos atributos que serán llamados en la clase Escena cada vez que se requieran. Además de disponer de algunos getters por si la escena necesitara poder acceder a alguno de estos atributos.Ahora vamos a los dos métodos más importantes de esta clase: *renderizar*() y *prerenderizar*() que se van a encargar de escribir nuestro objeto en la pantalla o en el FBO si lo requiere.

El método de renderizar necesita de un shader como entrada, además de una cámara para poder calcular la matriz de vista, un entero que nos indicara si vamos a tratar con una sombra, que tipo de sombra es y otros dos con el ancho y alto de la ventana. Con estos parámetros de entrada vamos a cargar las uniform correspondientes al shader para que pueda renderizar un objeto de tales características, una vez cargadas tendremos que ver que tipo de objeto es si un cubo o un plano y

en función de tal elegir un VAO u otro, dado que cuando cargamos los valores de los atributos en la clase al cargar el tipo de figura, automáticamente se cargarán los VAO con los vértices correspondientes, una vez elegido solo tendremos que dibujarlo y ejecutar glDrawArrays para que se renderice nuestra figura.

```
void Entity::renderizar(Shader shaderFig, glm::vec3 pos, Camera cam ,int sombra, int w, int h) {
    //Shader debug("VshaderDebug.vs", "FshaderDebug.fs");
    glm::mat4 projection = glm::perspective(glm::radians(cam.Zoom), (float)w / (float)h, 0.1f, 100.0f);
    glm::mat4 view = cam.GetViewMatrix();
    //if (tipoFigura == 0)
    //  matrizModelo = glm::rotate(matrizModelo, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
    glm::mat4 matrizModeloNormal = glm::transpose(glm::inverse(matrizModelo));

    shaderFig.use();
    shaderFig.setMat4("projection", projection);
    shaderFig.setMat4("view", view);
    shaderFig.setMat4("model", matrizModelo);
    shaderFig.setMat4("modelNorm", matrizModeloNormal);
    shaderFig.setInt("tipoTextura",tipoTextura);
    shaderFig.setInt("numTexturas", numTexturas);
    /*
    debug.use();
    debug.setInt("depthMap", 0);
    debug.setFloat("near_plane", 0.5);
    debug.setFloat("far_plane", 25.0);
    mapaSombra.vincularTextura(0);
    if (TBN == 0) {
        glBindVertexArray(VAO);
        if (tipoFigura == 0) {
            glDrawArrays(GL_TRIANGLES, 0, sizeof(vertices2) / 8);
        }
        else if (tipoFigura == 1) {
            glDrawArrays(GL_TRIANGLES, 0, sizeof(verticesQuad) / 8);
        }
        glBindVertexArray(0);
    }
```

Figura 23: Fragmento del método renderizar de la clase Entity.

```
        else {
            shaderFig.setInt("MapaDiff", 0);
            texturaMapaDiff.vincularTextura(0);
            shaderFig.setInt("MapaNormal", 1);
            texturaMapaNormal.vincularTextura(1);
            shaderFig.setInt("MapaProf", 2);
            mapaProfundidad.vincularTextura(2);
            if (sombra == 1) {
                shaderFig.setInt("MapaSombra", 3);
                mapaSombra.vincularTextura(3);
            }
            else if (sombra == 2) {
                shaderFig.setInt("CuboSombra", 3);
                cuboSombra.vincularCubo(3);
            }
            shaderFig.setInt("paralaje", 1);
            shaderFig.setFloat("hS", heigh_scale);
        }
        shaderFig.setInt("hacerTBN", 1);
    }

    if (TBN == 0) {
        glBindVertexArray(VAO);
        if (tipoFigura == 0) {
            glDrawArrays(GL_TRIANGLES, 0, sizeof(vertices2) / 8);
        }
        else if (tipoFigura == 1) {
            glDrawArrays(GL_TRIANGLES, 0, sizeof(verticesQuad) / 8);
        }
        glBindVertexArray(0);
    }
    else {
        glBindVertexArray(VAO);
        if (tipoFigura == 0)
            glDrawArrays(GL_TRIANGLES, 0, sizeof(verticesCuboTBN) / 14);
        else if (tipoFigura == 1)
            glDrawArrays(GL_TRIANGLES, 0, sizeof(verticesQuadTBN) / 14);
        glBindVertexArray(0);
    }
```

Figura 24: Fragmento del método renderizar de la clase Entity.

El método *prerenderizar*() es similar pero no cargará ninguna uniform y dibujara directamente la figura, ideal para shaders rápidos que no necesitan de ninguna uniform o ya se las han cargado de antes. Hemos hablado antes de que cuando cargamos los atributos a nuestro objeto cargamos el VAO con vértices correspondientes, y eso es posible porque nos hemos definido unas plantillas de vértices para cada tipo de figura, en especial un tipo de cuadrilatero que ocupa toda la pantalla que sirve para cuando usamos la técnica de la floración y la difuminación sobre uno de éstos para dar el resultado final. Pues bien existe un método de renderizado para que un objeto cualquiera dibuje este plano si se le pide, solo renderiza este plano especial.

Ahora que ya hemos visto la clase para construir un objeto no proveniente de un modelo artístico, para los otros que se cargarán con Assimp tenemos la siguiente clase que se ha tomado prestada de la página learnOpenGL y que la hemos adaptado para que sea compatible con nuestro motor gráfico:

b) **Model**: Para explicar esta clase solo nos vamos a limitar a explicar nuestras modificaciones y no lo que ya hemos tomado prestado, que podrán consultarlo aqui (Consulte aqui [37]). Para empezar hemos añadido el atributo modelMatrix que es la matriz modelo para poder ubicar nuestro personaje en el escenario y poder ajustarlo al tamaño correcto. Como nuevos métodos hemos añadido *Draw_mapaSombra*(), *Draw_cuboSombra*() y *Draw_normal*(), los dos primeros para poder sombrear nuestro modelo añadiendole las uniform necesarias y el tercero para añadirle las uniform cuando cargamos el mapaNormal ya que le faltaban algunas. También tenemos que tener en cuenta los métodos que hereda de ObjetoGráfico las cuales renderizar( ) se encargará de llamar a uno de los dos anteriores o al Draw original según se requiera y setModelMatrix( ) para cargar la matriz modelo. El resto de la clase es completamente igual que la original, también requiere del uso de Mesh.h cuyos metodos podemos consultarlos aquí [38].

2. **Luz**: Uno de los objetivos de este proyecto es aprender sobre técnicas de renderización de la iluminación de una escena, esta clase se encarga precisamente de eso. Tenemos una clase que se va a encargar de almacenar todas las características lumínicas de la escena y va a tratar de cargar en los shaders ciertas propiedades para que rendericen sus objetos con éstas. Como toda clase tenemos que empezar hablado de sus atributos y qué es lo que hacen. Empezamos también con un amplio listado:

   a) **focos**: Este atributo contiene una lista de structs *atrbLuzPuntual* que contienen las uniform que cargarle a cada foco de luz puntual, almacenan también la matriz de modelaje para renderizar el foco en la pantalla.

   b) **dirLuz**: Atributo de tipo struct *atrbLuzDireccional* que contiene los valores de las uniform que se cargarán en los shader para la luz direccional.

   c) **spotLuz**: Atributo de tipo struct *atrbLuzSpot* que contiene los valores de las uniform que se cargarán en los shader para la luz spot.

   d) **ligtVAO**/ **VBO**: Array de vértices cargado al inicializar la clase que contiene los vertices de los cubos blancos que se van a renderizar representando a los focos de luz puntual. Usarán las matrices de modelaje correspondiente a cada foco.

   e) **blinn**: Entero que sirve para indicar si la iluminación sigue el modelo de Blinn-Phong

   f) **tipoSombreado**: Entero que usamos para indicar que tipo de sombreado va a tener nuestra escena.

   g) **bloom**: Entero para indicar si la luz va a sufrir floración.

h) **hdrFBO**, junto a sus colorBuffers y su RBO de profundidad: FBO que vamos a utilizar para renderizar en él la escena normal y la escena con los tonos luminosos para que estos últimos sean difuminados y podamos crear el efecto de floración.

i) **bluringFBO[ ]** junto a sus colorBuffers respectivos: Dos FBOs que se van a ir mandando los resultados de sus renderizaciones entre sí hasta terminar las iteraciones de la fase de difuminación.

j) **lastBuff**: Índice que indica cual de los dos FBOs anteriores contiene el resultado final.

Ahora que hemos visto que parámetros va a gestionar esta clase veamos como los emplea en sus métodos. Primero como siempre toca hablar de los métodos de carga de atributos llamados desde la escena cuando ésta los carga de los archivos, también los de inicialización que se van a encargar de generar los FBOs correspondientes y de cargar el VAO de los focos. Pero los importantes son los que se usan en la etapa de renderización de los cuales destacamos los siguientes:

a) **renderizarLuz( )**: Este método recibe como parámetros 3 shaders, dos que son los shaders de las figuras y los modelos de assimp, el otro es el shader encargado de renderizar las figuras que representan a los focos de luz puntual. También recibe la cámara para que cargue la posición de ésta, un entero para indicar si se está sombreando la escena o no y por último el ancho y altura de la ventana. Con estos parámetros y los atributos de esta clase, este método se va a encargar de cargar las uniform de los 3 shaders correspondientes para que estos puedan ejecutarse y dar los resultados correctos.

```
void Luz::renderizarLuz(Shader shaderFig, Shader shaderMod , Shader Shaderfocos, glm::vec3 pos, Camera cam, int sombra, int w, int h) {
    shaderFig.use();
    if (sombra == 0) {
        //Configuramos la camara
        shaderFig.setVec3("PosVista", pos);
        //Configuramos los atributos de la luz direccional.
        shaderFig.setVec3("luzDir.direccion", dirLuz.dir);
        shaderFig.setVec3("luzDir.ambiente", dirLuz.ambiente);
        shaderFig.setVec3("luzDir.difusion", dirLuz.diffusion);
        shaderFig.setVec3("luzDir.especular", dirLuz.especular);
        //Configuramos los atributos de la luz de spot
        shaderFig.setVec3("luzSpot.posicion", cam.Position);
        shaderFig.setVec3("luzSpot.direccion", cam.Front);
        shaderFig.setVec3("luzSpot.ambiente", spotLuz.ambiente);
        shaderFig.setVec3("luzSpot.difusion", spotLuz.diffusion);
        shaderFig.setVec3("luzSpot.especular", spotLuz.especular);
        shaderFig.setFloat("luzSpot.constante", spotLuz.k);
        shaderFig.setFloat("luzSpot.lineal", spotLuz.lin);
        shaderFig.setFloat("luzSpot.cuadratica", spotLuz.q);
        shaderFig.setFloat("luzSpot.cOff", spotLuz.cOff);
        shaderFig.setFloat("luzSpot.ocOff", spotLuz.ocOff);
        //Configuramos los atributos de cada foco
        int num = 0;
        for (atrbLuzPuntual luzPunt : focos) {
            string tipo = "luzFoco[" + to_string(num);
            shaderFig.setVec3(tipo + "].posicion", luzPunt.pos);
            shaderFig.setVec3(tipo + "].ambiente", luzPunt.ambiente);
            shaderFig.setVec3(tipo + "].difusion", luzPunt.diffusion);
            shaderFig.setVec3(tipo + "].especular", luzPunt.especular);
            shaderFig.setFloat(tipo + "].constante", luzPunt.k);
            shaderFig.setFloat(tipo + "].lineal", luzPunt.lin);
            shaderFig.setFloat(tipo + "].cuadratica", luzPunt.q);
            num++;
        }
        shaderFig.setInt("nFocos", focos.size());
        shaderFig.use();
        shaderFig.setInt("Sombreado", 0);
    }
```

```
        else {
            string tipo = "luzFoco[0";
            shaderMod.setVec3(tipo + "].posicion", focos.at(0).pos);
            shaderMod.setVec3(tipo + "].ambiente", focos.at(0).ambiente);
            shaderMod.setVec3(tipo + "].difusion", focos.at(0).diffusion);
            shaderMod.setVec3(tipo + "].especular", focos.at(0).especular);
            shaderMod.setFloat(tipo + "].constante", focos.at(0).k);
            shaderMod.setFloat(tipo + "].lineal", focos.at(0).lin);
            shaderMod.setFloat(tipo + "].cuadratica", focos.at(0).q);
            shaderMod.setInt("nFocos", focos.size());
        }
        shaderMod.use();
        shaderMod.setInt("Sombreado", 1);
    }
    //Configuramos el modelo de iluminacion
    shaderMod.setInt("blinn", blinn);


    glm::mat4 projection = glm::perspective(glm::radians(cam.Zoom), (float)w / (float)h, 0.1f, 100.0f);
    glm::mat4 view = cam.GetViewMatrix();

    Shaderfocos.use();
    Shaderfocos.setMat4("projection", projection);
    Shaderfocos.setMat4("view", view);
}
```

Figura 25: Fragmento del método renderizar de la clase Luz.

b) **renderizarBlur( )**: Método encargado de renderizar continuamente entre los dos FBOs de difuminación la textura de las zonas iluminadas de la escena, para generar su textura difusa. Recibe como parámetros el shader encargado de difuminar para cargarle sus uniforms y ponerlo en marcha. Además requiere de un objeto entity cualquiera para que renderice el cuadrilátero de pantalla que se renderizará para generar la textura buscada. Este método solo será llamado si se está utilizando el efecto de floración de la luz.

```
void Luz::renderizarBlur(Shader blurr, Entity figura) {
    bool horizontal = 1, primero = true;
    unsigned int difuminacion = 15;
    blurr.use();
    blurr.setInt("imagenParaModificar", 0);
    for (unsigned int i = 0; i < difuminacion; i++)
    {
        glBindFramebuffer(GL_FRAMEBUFFER, bluringFBO[horizontal]);
        blurr.setBool("horizontal", horizontal);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, primero ? colorBuffers[1] : bluringColorbuffers[!horizontal]);
        figura.renderizaQuad();
        horizontal = !horizontal;
        if (primero)
            primero = false;
    }
    lastBuff = horizontal;
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Figura 26: Método renderizarBlur de la clase Luz.

c) **renderizarBloom( )**: Método encargado de renderizar la escena final con floración en la pantalla, tendrá que recibir el shaderBloom de la escena para que le cargue las uniforms y pueda ejecutarlo. Este método solo será llamado si se está utilizando el efecto de floración de la luz.

```
void Luz::renderizarBloom(Shader blooming, Entity figura) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    blooming.use();
    blooming.setInt("escena", 0);
    blooming.setInt("bluringCompleto", 1);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, colorBuffers[0]);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, bluringColorbuffers[!lastBuff]);
    figura.renderizaQuad();
}
```

Figura 27: Método renderizarBloom de la clase Luz.

d) **abrir**/ **cerrarFBO( )**: Métodos encargados de vincular y cerrar respectivamente el FBO encargado de renderizar en dos texturas la textura de la escena original y la de los tonos más lumínicos de ésta. Estos métodos solo serán llamados si se está utilizando el efecto de floración de la luz.

```
void  Luz::abrirFBF() {
    glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
void  Luz::cerrarFBF() {
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Figura 28: Métodos para abrir y cerrar el FBO de la clase Luz.

e) **renderizaFocos( )**: Método que recibe el shader que renderiza las figuras de los focos de luz puntual, le carga la matriz modelo de cada foco y lo ejecuta para que se renderice en pantalla.

```
void Luz::renderizaFocos(Shader shad, Camera cam, int w , int h) {

    for (atrbLuzPuntual foco : focos) {
        shad.use();
        shad.setMat4("model", foco.modelo);

        glBindVertexArray(lightVAO);
            glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
    }
}
```

Figura 29: Método para renderizar los focos de la escena.

Con esto terminamos la primera parte de la iluminación, la segunda reside en la siguiente clase que se encarga de generar las texturas de sombra en función de la iluminación.

3. **Sombra**: Esta clase va a gestionar la generación de sombras de la escena y por tanto va a tener que renderizar ésta previamente generando una textura con las sombras buscadas, es por eso que vamos a tener que guardar atributos que gestionen estos procedimientos de los cuales podemos listar los siguientes:

   a) **tipoSombra**: Entero que indica el tipo de sombra que tenemos, 0 si es solamente de luz direccional y 1 si se trata de luz posicional.

   b) **posLuz**: Atributo de tipo vec3 que indica la posición donde vamos a situar el foco de luz, en caso de que sea direccional cogeremos la dirección y la pasaremos negada multiplicada por un entero, para que el efecto sea adecuado a la iluminación de la escena.

   c) **planoCercano** y **planoLejano**: Valores del plano cercano y lejano en los que vamos a proyectar la escena cuando la rendericemos en el FBO.

   d) **mapaSombraProfundidadFBO**, **mapaSombraProfundidad** y **cuboSombraProfundidad**: El primer atributo es el FBO que va a renderizar la escena y generar la textura de sombra que se usará para renderizar la escena final, el segundo es el identificador de esa textura generada, y por último el tercero es la textura cúbica generada en caso de que estemos tratando con luces puntuales.

   e) **matrizLuz**: Atributo de tipo mat4 que se va a encargar de generar una matriz que sitúe a los puntos en el espacio que tenga el foco de luz por punto de referencia. Este se deberá cargar en el shader de sombreado y en el shader final para que mueva los fragmentos que necesiten saber su orientación en ese espacio.

   Una vez vistos los atributos que maneja la clase podemos dar el paso y explicar como los usamos en los métodos para poder renderizar nuestra escena en la textura, como toda clase tenemos métodos de carga e inicialización de atributos, los cuales tenemos uno para inicializar los datos aportados por el archivo de la luz y otro para inicializar el FBO que va renderizar la escena en la textura de la sombra, o el cubo si es el caso. También tenemos métodos para cargar en la escena la textura sombra generada tras la renderización, así como el cubo de sombras. Y por último tenemos el método más importante que es el de la renderización.

```
void Sombra::renderizaSombra(Shader Preshad, Shader PostshadFig, Shader PostshadMod, Camera cam,vector<Entity> figuras,
    vector<Model> modelos, int widt, int hight) {
    glViewport(0, 0, widt, hight);
    glBindFramebuffer(GL_FRAMEBUFFER, mapaSombraProfundidadFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    if (tipoSombra == 0) {
        glm::mat4 proyeccionLuz, vistaLuz;
        proyeccionLuz = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, planoCercano, planoLejano);
        vistaLuz = glm::lookAt(posLuz, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
        matrizLuz = proyeccionLuz * vistaLuz;
        Preshad.use();
        Preshad.setMat4("lightSpaceMatrix", matrizLuz);
    }
    else { //Ahora nos vamos al caso de que renderizamos un mapeado por cubo
        glm::mat4 proyeccionSombra = glm::perspective(glm::radians(90.0f), (float)widt / (float)hight, planoCercano, planoLejano);
        std::vector<glm::mat4> matricesSombra;
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(-1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(0.0, 1.0, 0.0), glm::vec3(0.0, 0.0, 1.0)));
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(0.0, -1.0, 0.0), glm::vec3(0.0, 0.0, -1.0)));
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(0.0, 0.0, 1.0), glm::vec3(0.0, -1.0, 0.0)));
        matricesSombra.push_back(proyeccionSombra *
            glm::lookAt(posLuz, posLuz + glm::vec3(0.0, 0.0, -1.0), glm::vec3(0.0, -1.0, 0.0)));
        Preshad.use();
        for (unsigned int j = 0; j < 6; j++) {
            Preshad.setMat4("sombraMatriz[" + to_string(j) + "]", matricesSombra.at(j));
        }
        Preshad.setFloat("planoLejano", planoLejano);
        Preshad.setVec3("posLuz", posLuz);
    }
```

Figura 30: Fragmento del método para renderizar las sombras en una textura.

Este método recibe como parámetros los shaders que renderizan las figuras, los modelos y también el shader que se va a encargar de generar la textura de la sombra, o del cubo de sombras. Para poder renderizar la escena tiene que recibir tanto la cámara de la escena como los objetos, figuras y modelos Assimp de la misma escena ya que tiene que pasarlos por el shader para que éste los renderice en el FBO y los dibuje en la textura o cubo de sombras, también como todo método de renderizado, necesitará las dimensiones de la ventana de openGL. Este método renderizará en el FBO los objetos de la escena y generará la nueva textura o cubo de sombras, posteriormente cargará en los shaders de los modelos y de las figuras las uniforms correspondientes para que puedan generar la escena final con las sombras bien dibujadas.

4. **MapaCubo**: Esta clase la vamos a utilizar para gestionar las texturas cúbicas, es decir las formadas por 6 texturas 2D que forman un cubo, en nuestro caso vamos a utilizar estas texturas para generar el fondo de nuestra escena. Por lo que esta clase tendrá pocos atributos y métodos con los que gestionar sus renderizados. Solo tendremos 3 atributos, dos para el VAO y el VBO donde se cargarán los vértices del fondo para ser renderizados y otro atributo para un identificador de textura cúbica.

Tendremos cuatro métodos para esta clase, uno para vincular en caso de que tratemos con una textura cúbica y otro para inicializar el fondo del escenario cargando los vértices en el VAO. También tendremos un metodo para cargar una textura cúbica desde un archivo, este método lo hemos tomado prestado de los manuales de openGL (Vease en [40]). Por último tenemos otro método para renderizar sobre la pantalla

el fondo de la escena o la textura cúbica que tengamos asociada, se completarán las uniform en el shader que le pasemos por parámetro y éste al ejecutarse renderizará el mapa cúbico correspondiente.

```
void mapaCubo::cargarTexturas(vector<string> caras) {
    glGenTextures(1, &idCubo);
    glBindTexture(GL_TEXTURE_CUBE_MAP, idCubo);

    int w, h, nC;
    for (unsigned int i = 0; i < caras.size(); i++)
    {
        unsigned char *data = stbi_load(caras[i].c_str(), &w, &h, &nC, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Fallo al cargar una de las caras: " << caras[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
}
```

Figura 31: Método para cargar el cubo de seis archivos.

5. **Textura**: Para poder tratar todas las texturas que utilizamos en nuestro motor vamos a crear una clase Textura para que realice tanto las funciones de carga como la vinculación de las mismas. Para ello nos bastamos con un único atributo *idTextura* que será el valor donde crearemos la textura asociada a la clase.

   Dispondremos de un método de carga de texturas por archivos basado en el método visto en los tutoriales (Vease en [41]), una vez tengamos creada y cargada la textura deseada dispondremos de dos métodos para invertir la imagen si esta no nos viniera en el formato adecuado, hay texturas que vienen al revés, y por último un método para vincular la textura con el shader antes de ejecutarlo para que pueda acceder a ésta, en este método podremos elegir la posición de textura donde queramos vincularla.

```
void Textura :: cargarTextura(char * path) {
    glGenTextures(1, &idTextura);
    int width, height, nrComponents;
    unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, idTextura);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        //Cambiar para ser configurable en la version definitiva :)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        cout << "Error en la carga de la textura." << std::endl;
        stbi_image_free(data);
    }
}
```

Figura 32: Método para cargar la textura de un archivo.

```
void Textura::vincularTextura(int i) {
    if (i == 0) {
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, idTextura);
    }
    else if (i == 1) {
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, idTextura);
    }
    else if(i == 2) {
        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, idTextura);
    }
    else if (i == 3) {
        glActiveTexture(GL_TEXTURE3);
        glBindTexture(GL_TEXTURE_2D, idTextura);
    }
    else {
        glActiveTexture(GL_TEXTURE4);
        glBindTexture(GL_TEXTURE_2D, idTextura);
    }
}
```

Figura 33: Método para vincular la textura al shader.

6. **Particula**: En la sección 3.7 definimos el shader de partículas basado en lo que hace

exactamente la versión de nuestro shader, luego asociaremos rápido los conceptos con los que esta clase trata. Esta clase va a disponer de los siguientes atributos:

a) **datos**: Atributo de clase struct *ParticulaIndividual* que está formado por la posición inicial, la velocidad inicial y el tiempo de vida de toda partícula.

b) **tiempoParticulas**: Vector que contiene el tiempo de inicio de todas las partículas que han sido creadas hasta el momento. Cada vez que se cree una partícula, se añadirá a este vector el tiempo en el que se haya creado.

c) **matrizModelo**: Matriz de modelaje que van a seguir todas las partículas que se ejecuten en el shader.

d) **gravity**: Valor Float que indica la aceleración que va a tener la partícula a lo largo del tiempo, si es negativa simulará un efecto de gravedad.

e) **ParticlesVAO**/ **VBO**: VAO y VBO encargados de cargar la posición inicial, la velocidad y el tiempo de vida de las partículas, a cada ejecución del shader se accederá a estos valores. Se cargarán en el VAO dentro del método de carga de éstos.

f) **NUMPART** / **numParticulasCreadas**: Número máximo de partículas a crear en la escena, y número de partículas creadas en el momento.

g) **tex**: Textura que van a tomar las partículas cuando se rendericen en pantalla, podemos introducir la textura que simule el efecto deseado, en nuestro caso más destacado crearemos humo ayudándonos de texturas de éste.

Una vez vistos los atributos donde se almacena la información de nuestro sistema de partículas, podemos destacar los métodos de carga de todos los atributos de la clase que en este caso son tres, uno de carga de los parámetros iniciales de las partículas, la aceleración y el número de partículas que se creará en el sistema, en este método se cargará el VAO con los datos iniciales de las partículas, otro método para cargar la matriz modelo y el otro para cargar la textura que definirá a nuestras partículas. El último método es el de renderizado que traerá como parámetros el shader que gestiona los sistemas de partículas, la cámara de la escena, el tiempo actual y las dimensiones de la ventana. En este método rellenaremos las uniforms correspondientes y ejecutaremos el shader las veces que partículas hayamos creado hasta el momento. En cada iteración del bucle, si no se han creado todas, se irá creando una nueva partícula.

```
void Particulas::renderizarParticulas(Shader shaderParticle, Camera cam, float tiempo, int w, int h) {

    glm::mat4 projection = glm::perspective(glm::radians(cam.Zoom), (float)w / (float)h, 0.1f, 100.0f);
    glm::mat4 view = cam.GetViewMatrix();
    shaderParticle.use();
    shaderParticle.setFloat("tiempo", tiempo);
    shaderParticle.setVec3("acc", glm::vec3(gravity, gravity, gravity));
    shaderParticle.setMat4("projection", projection);
    shaderParticle.setMat4("view", view);
    shaderParticle.setMat4("model", matrizModelo);
    shaderParticle.setInt("textura",0);
    tex.vincularTextura(0);

    //Si no han salido todas sacamos una...
    if (numParticulasCreadas < NUM_PART) {
        tiempoParticulas.push_back(tiempo);
        shaderParticle.setFloat("t_ini", tiempo);
        glBindVertexArray(ParticlesVAO);
            glDrawArrays(GL_POINTS, 0, 7);
        glBindVertexArray(0);
    }

    for (int i = numParticulasCreadas - 1; i >= 0; i--) {
        shaderParticle.setFloat("t_ini", tiempoParticulas.at(i));
        glBindVertexArray(ParticlesVAO);
            glDrawArrays(GL_POINTS, 0, 7);
        glBindVertexArray(0);
    }

    numParticulasCreadas++;

}
```

Figura 34: Método para renderizar las partículas de la escena.

7. **Camera**: Esta clase es la encargada de generar la matriz de vista que hace que los objetos que rendericemos se muevan a la referencia de nuestra vista. Además es la encargada de gestionar todos los movimientos de cámara que se encargarán de modificar nuestra matriz de vista. La implementación de la clase la hemos tomado prestada también de los manuales debido a la gran completitud que estos le aportan y hace que nos sirva para poder observar a la perfección la escena que estamos renderizando, (vease en [39]). La clase dispondrá de los métodos encargados de recalcular nuestra cámara según la dirección en la que estemos moviendo, girando o acercando ésta.

   Estos serán llamados por la clase Escenario cada vez que haya un nuevo input que involucre un movimiento de cámara. Cabe recordar que en GraphicProgram tenemos los métodos que llaman a los de escena y además en el Main tenemos algunos otros métodos que se activarán en cada interrupción que se genere al mover o scrollear el ratón, llamando a los que están en GraphicProgram, creando una serie de llamadas hasta llegar a los de esta clase, que son los que hacen verdaderamente estas operaciones.

   En estos métodos destacamos ProcessKeyboard( ) que se encarga de mover la cámara en la dirección que le indiquemos con el teclado, ProcessMouseMovement( ) que girará la cámara en función de la posición anterior y la posición actual a la que se haya movido, y otro que es ProcessMouseScroll( ) para calcular el Zoom de la cámara en función de la entrada que le llegue del Scroll. Luego tenemos el método GetViewMatrix( ) para acceder a la matriz de vista cada vez que la necesitemos. Con estos métodos podemos resumir perfectamente el funcionamiento de esta clase.

Una vez vistas todas las clases utilizadas en el programa nos toca abordar los nueve shaders que hemos utilizado para poder renderizar nuestra escena. Ya que estos son los que verdaderamente van a ejecutar las operaciones en nuestra GPU y generar así los gráficos de nuestro pequeño motor.

4.2. Shaders que utilizamos para el renderizado

En esta sección nos vamos a dedicar a explicar el funcionamiento de los shaders que utilizamos en nuestro programa, trataremos de describir todas las funciones que éstos utilizan, las entradas y salidas que éstos reciben/generan y el significado de las uniforms para controlar su funcionamiento. Empezamos por el más importante, el encargado de renderizar las figuras de la escena.

1. **ShaderFigura**: Como bien hemos dicho antes todas las figuras que no sean modelos importados con Assimp van a pasar por este shader para poder ser renderizados y pasados a pantalla. Tenemos un shader formado por un shader de vértices y otro de fragmentos. Empezamos a explicarlos por orden de ejecución.

   El shader de vértices tendrá 5 posibles entradas del buffer: aPos para recibir los vértices, aNorm para las normales, aTC para las coordenadas de textura, aTan para las tangentes y aBiTan para las binormales. Como uniforms tendremos muchas, que en función de las uniforms de control se usarán unas u otras, el objetivo es darle al shader la funcionalidad en el mayor número de casos de escenario posible. De estas uniforms de control podremos considerar las siguientes:

   a) **Sombreado**: Valor entero que indica si la figura tiene que tener en cuenta las sombras de la escena.

   b) **TipoSombreado**: Valor entero que indica qué tipo de sombreado se va a aplicar a la figura.

   c) **hacerTBN**: Valor entero que indica si la figura tiene que cargar tangentes y binormales.

   Ahora podemos hablar de las uniforms que van a ser utilizadas para hacer los cálculos las cuales destacamos las siguientes:

   a) **projection**: Matriz de proyección por la que van a pasar los puntos del espacio visual.

   b) **model**: Matriz de modelaje para situar nuestra figura en el espacio.

   c) **view**: Matriz de vista para mover los puntos hacia la cámara.

   d) **modelNorm**: Matriz para modelar las normales de los vértices.

   e) **lightSpaceMatrix**: Matriz que mueve la posición de un fragmento para situarlo dentro del espacio de visión de la luz.

En caso de que no usemos tangentes en nuestro buffer de entrada, calcularemos la posición del fragmento pasándolo por la matriz de modelaje que irá como salida al siguiente shader, lo mismo con las normales que las pasaremos por la modelNorm que se encarga de modelar este tipo de vectores. Las coordenadas no tendremos que transformarlas y las pasaremos directamente, en caso de tener sombreado con luces direccionales tendremos que calcular la posición del fragmento en el espacio con la luz como punto de referencia, pasando el fragmento por la matriz lightSpaceMatrix, por último calcularemos glPosition pasando la posición por el modelaje, la vista y la proyección.

En el caso de que si usemos tangentes tendremos que calcular la normal, tangente y binormal preparadas en el espacio modelado, es decir, pasarlas por modelNorm. Una vez con estos vectores formaremos la matriz TBN, y con esta calcularemos la posición del fragmento, que deberá estar modelada, en el espacio tangente pasándole la matriz TBN. Lo mismo deberemos hacer con la posición de la luz y la de la cámara. Una vez hecho esto, como en el otro caso calculamos glPosition de la misma forma y la pasamos al shader de fragmentos.

```
void main () {
    if (hacerTBN == 0) {
        posFragmento = vec3(model * vec4(aPos, 1.0));
        normal = mat3(modelNorm) * aNorm;
        TC = aTC;
        if (Sombreado == 1 && TipoSombreado == 0) {
            posFragmentoEspacioLuz = lightSpaceMatrix * vec4(posFragmento, 1.0);
        }
        gl_Position = projection * view * vec4(posFragmento, 1.0);
    } else {
        posFragmento = vec3(model * vec4(aPos, 1.0));
        TC = aTC;

        vec3 T = normalize(mat3(modelNorm) * aTan);
        vec3 N = normalize(mat3(modelNorm) * aNorm);
        vec3 B = normalize(mat3(modelNorm) * aBiTan);

        TBN = transpose(mat3(T, B, N));
        PosicionFragmentoTangente  = TBN * posFragmento;
        if (Sombreado == 1 && TipoSombreado == 0) {
            posFragmentoEspacioLuz = lightSpaceMatrix * vec4(posFragmento, 1.0);
        }
        gl_Position = projection * view * model * vec4(aPos, 1.0);
    }
}
```

Figura 35: Shader de vértices de una figura.

Ahora vamos con la parte más compleja de este shader, el de fragmentos. Aquí tendremos que describir las uniform utilizadas, los métodos para calcular los distintos tipos de luz, el paralaje y las sombras, por último vamos a ver como se ejecuta el shader en función de las uniforms de control. Empezamos hablando de las uniforms:

a) **blinn**: Entero que indica si el shader usará el modelo de Bling-Phong a la hora de tratar la iluminación.

b) **tipoTextura**: Valor entero que indica si la textura la cargará de un tipo de material o cargará las texturas directamente sin ir asociadas a un material.

c) **numTexturas**: Indica el número de texturas que va utilizar nuestra figura.

d) **hacerTBN**: Indica si los vértices se cargaron con las tangentes y binormales.

e) **paralaje**: Indica si la figura utiliza mapeo con paralaje.

f) **hS**: Valor entero de control de paralaje, para que el offset no se pase demasiado.

g) **Sombreado**: Entero que indica si tenemos que pintar las sombras en esta figura.

h) **TipoSombreado**: Entero que indica que tipo de sombra hay que dibujarle en caso de que hubiera que hacerlo.

Ahora después de conocer las uniforms con las que controlaremos la ejecución, veremos las que usamos para emplear los cálculos de nuestro fragmento:

a) **MapaDiff**/ **MapaNormal**/ **MapaProf**: Texturas asociadas a la textura difusa, la textura normal y la textura de profundidad.

b) **MapaSombra** y **CuboSombra**: Textura asociada al mapa de sombras y Textura cúbica asociada al mapa cúbico de sombras de la escena.

c) **PosVista**: Posición donde se sitúa el observador/cámara en la escena.

d) **luzDir**: Struct con los parámetros, vistos en la sección anterior, de la luz de tipo direccional.

e) **luzFoco**: Struct con los parámetros, vistos en la sección anterior, de la luz de tipo puntual.

f) **luzSpot**: Struct con los parámetros, vistos en la sección anterior, de la luz de tipo spot.

g) **Mater**: Struct que contiene las texturas difusa y especular de un material y su brillo asociado.

h) **textura1**/ **2**/ **3**: Texturas libres que se cargan en la figura si tipoTextura es 0 y no hay mapeado por normales ni por paralaje, es decir hacerTBN es 0.

Una vez vistas las uniforms veremos las funciones que utilizamos para calcular los resultados que nos aportan los tipos de luces, el mapeo por paralaje, o el sombreado de los objetos:

a) **generaLuzDir**/ **2**/ **TBN( )**: Funciones encargadas de calcular la luz direccional que afectará a nuestra figura, recibe como entrada los parámetros de la luz, la normal y la posición del observador. Para calcular la componente especular usará el vector reflejado o el vector medio en función del tipo de modelo que usemos. En el caso de que usemos la función para tangentes se usarán las posiciones convertidas en el shader de vértices con la matriz TBN.

```
vec3 generaLuzDir(LuzDireccional luz, vec3 n, vec3 dirVista) {
    vec3 dL = normalize(-luz.direccion);

    float diff = max(dot(n, dL), 0.0);
    float spec = 0.0;
    if(blinn == 0) {
        vec3 luzRef = reflect(-dL, n);
        spec = pow(max(dot(dirVista, luzRef), 0.0), Mater.brillo);
    }
    else {
        vec3 halfwayDir = normalize(dL + dirVista);
        spec = pow(max(dot(n, halfwayDir), 0.0), Mater.brillo);
    }
    vec3 ambiente = luz.ambiente * vec3(texture(Mater.texDiff1, TC));
    vec3 difusion = luz.difusion * diff * vec3(texture(Mater.texDiff1, TC));
    vec3 especular = luz.especular * spec * vec3(texture(Mater.texOp, TC));
    float shade = 0.0;
    if (Sombreado == 1 && TipoSombreado == 0) {
        shade = calculaSombraNoCubo(posFragmentoEspacioLuz);
    }
    return (ambiente + (1.0 - shade) * (difusion + especular));
}
```

```
vec3 generaLuzDirTBN(LuzDireccional luz, vec3 n, vec3 TanDirVista) {
    vec2 coordTexturas = TC;
    if (paralaje == 1) {
        coordTexturas = ParallaxMapping(TC,TanDirVista);
    }

    vec3 dL = normalize(-TBN*luz.direccion);

    float diff = max(dot(n, dL), 0.0);

    float spec = 0.0;
    if(blinn == 0) {
        vec3 luzRef = reflect(-dL, n);
        spec = pow(max(dot(TanDirVista, luzRef), 0.0), 32.0);
    }
    else {
        vec3 halfwayDir = normalize(dL + TanDirVista);
        spec = pow(max(dot(n, halfwayDir), 0.0), 32.0);
    }

    vec3 ambiente = luz.ambiente * vec3(texture(MapaDiff, coordTexturas));
    vec3 difusion = luz.difusion * diff * vec3(texture(MapaDiff, coordTexturas));
    vec3 especular = luz.especular * spec;
    float shade = 0.0;
    if (Sombreado == 1 && TipoSombreado == 0) {
        shade = calculaSombraNoCubo(posFragmentoEspacioLuz);
    }
    return (ambiente + (1.0 - shade) * (difusion + especular));
}
```

Figura 36: Shader de fragmentos, función para la luz direccional.

b) **generaLuzPuntual/ 2/ TBN( )**: Funciones encargadas de calcular la luz puntual que afectará a nuestra figura, recibe los parámetros de las funciones anteriores más la posición del fragmento y los parámetros de la luz esta vez son los de la direccional. Al igual que las anteriores calculará la componente especular en función del modelo y en caso de que sea el método para tangentes se pasará la posición del fragmento en el espacio tangente al igual que se usarán las posiciones correspondientes de este espacio.

```
vec3 generaLuzPuntual(LuzPuntual luz, vec3 n, vec3 posFrag, vec3 dirVista) {
    vec3 dL = normalize(luz.posicion - posFrag);

    float diff = max(dot(n, dL), 0.0);

    float spec = 0.0;
    if(blinn == 0) {
    |   vec3 luzRef = reflect(-dL, n);
    |   spec = pow(max(dot(dirVista, luzRef), 0.0), Mater.brillo);
    }
    else {
    |   vec3 halfwayDir = normalize(dL + dirVista);
    |   spec = pow(max(dot(n, halfwayDir), 0.0), Mater.brillo);
    }

    float dist = length(luz.posicion - posFrag);
    float at = 1.0 / (luz.constante + luz.lineal * dist + luz.cuadratica * (dist * dist));

    vec3 ambiente = luz.ambiente * vec3(texture(Mater.texDiff1, TC));
    vec3 difusion = luz.difusion * diff * vec3(texture(Mater.texDiff1, TC));
    vec3 especular = luz.especular * spec * vec3(texture(Mater.texOp, TC));
    ambiente *= at;
    difusion *= at;
    especular *= at;
    float shade = 0.0;
    if (Sombreado == 1 && TipoSombreado == 1) {
    |   shade = calculaSombraCubo(posFrag);
    }
    return (ambiente + (1.0 - shade) * (difusion + especular));
}
vec3 generaLuzPuntualTBN(LuzPuntual luz, vec3 n, vec3 PosicionFragmentoTangente, vec3 TanDirVista){
    vec2 coordTexturas = TC;
    if (paralaje == 1) {
    |   coordTexturas = ParallaxMapping(TC,TanDirVista);
    }

    vec3 dL = normalize(TBN*luz.posicion - PosicionFragmentoTangente);

    float diff = max(dot(n, dL), 0.0);

    float spec = 0.0;
    if(blinn == 0) {
    |   vec3 luzRef = reflect(-dL, n);
    |   spec = pow(max(dot(TanDirVista, luzRef), 0.0), 32.0);
    }
    else {
    |   vec3 halfwayDir = normalize(dL + TanDirVista);
    |   spec = pow(max(dot(n, halfwayDir), 0.0), 32.0);
    }

    float dist = length(TBN*luz.posicion - PosicionFragmentoTangente);
    float at = 1.0 / (luz.constante + luz.lineal * dist + luz.cuadratica * (dist * dist));

    vec3 ambiente = luz.ambiente * vec3(texture(MapaDiff, coordTexturas));
    vec3 difusion = luz.difusion * diff * vec3(texture(MapaDiff, coordTexturas));
    vec3 especular = luz.especular * spec;
    ambiente *= at;
    difusion *= at;
    especular *= at;
    float shade = 0.0;
    if (Sombreado == 1 && TipoSombreado == 1) {
    |   shade = calculaSombraCubo(PosicionFragmentoTangente);
    }
    return (ambiente + (1.0 - shade) * (difusion + especular));
}
```

Figura 37: Shader de fragmentos, funciones para la luz puntual.

c) **generaLuzSpot/ 2/ TBN( )**: Similar a las funciones de luz puntual, mismos parámetros salvo que en vez de direccional son parámetros de luz spot, y los cálculos se harán respecto a este tipo de luz.

```
vec3 generaLuzSpot(LuzSpot luz, vec3 n, vec3 posFrag, vec3 dirVista) {
    vec3 dL = normalize(luz.posicion - posFrag);

    float diff = max(dot(n, dL), 0.0);

    float spec = 0.0;
    if(blinn == 0) {
    |   vec3 luzRef = reflect(-dL, n);
    |   spec = pow(max(dot(dirVista, luzRef), 0.0), Mater.brillo);
    }
    else {
    |   vec3 halfwayDir = normalize(dL + dirVista);
    |   spec = pow(max(dot(n, halfwayDir), 0.0), Mater.brillo);
    }

    float dist = length(luz.posicion - posFrag);
    float at = 1.0 / (luz.constante + luz.lineal * dist + luz.cuadratica * (dist * dist));

    float theta = dot(dL, normalize(-luz.direccion));
    float epsilon = luz.cOff - luz.ocOff;
    float intensidad = clamp((theta - luz.ocOff) / epsilon, 0.0, 1.0);

    vec3 ambiente = luz.ambiente * vec3(texture(Mater.texDiff1, TC));
    vec3 difusion = luz.difusion * diff * vec3(texture(Mater.texDiff1, TC));
    vec3 especular = luz.especular * spec * vec3(texture(Mater.texOp, TC));
    ambiente *= at * intensidad;
    difusion *= at * intensidad;
    especular *= at * intensidad;
    return (ambiente + difusion + especular);
}
```

```
vec3 generaLuzSpotTBN(LuzSpot luz, vec3 n, vec3 PosicionFragmentoTangente, vec3 TanDirVista){
    vec2 coordTexturas = TC;
    if (paralaje == 1) {
    |   coordTexturas = ParallaxMapping(TC,TanDirVista);
    }

    vec3 dL = normalize(TBN*luz.posicion - PosicionFragmentoTangente);

    float diff = max(dot(n, dL), 0.0);

    float spec = 0.0;
    if(blinn == 0) {
    |   vec3 luzRef = reflect(-dL, n);
    |   spec = pow(max(dot(TanDirVista, luzRef), 0.0), 32.0);
    }
    else {
    |   vec3 halfwayDir = normalize(dL + TanDirVista);
    |   spec = pow(max(dot(n, halfwayDir), 0.0), 32.0);
    }

    float dist = length(TBN*luz.posicion - PosicionFragmentoTangente);
    float at = 1.0 / (luz.constante + luz.lineal * dist + luz.cuadratica * (dist * dist));

    float theta = dot(dL, normalize(-TBN*luz.direccion));
    float epsilon = luz.cOff - luz.ocOff;
    float intensidad = clamp((theta - luz.ocOff) / epsilon, 0.0, 1.0);

    vec3 ambiente = luz.ambiente * vec3(texture(MapaDiff, coordTexturas));
    vec3 difusion = luz.difusion * diff * vec3(texture(MapaDiff, coordTexturas));
    vec3 especular = luz.especular * spec;
    ambiente *= at * intensidad;
    difusion *= at * intensidad;
    especular *= at * intensidad;
    return (ambiente + difusion + especular);
}
```

Figura 38: Shader de fragmentos, funciones para la luz spot.

d) **ParallaxMapping( )**: Función encargada de calcular la posición de la textura ajustada al paralaje por pasos con interpolación, recibe como valores la posición del fragmento en el espacio tangente.

```
vec2 ParallaxMapping(vec2 tc, vec3 vd) {

    float numCapas = mix(32, 8, abs(dot(vec3(0.0, 0.0, 1.0), vd)));
    float profCapa = 1.0 / numCapas;
    float capaActual = 0.0;
    vec2 P = vd.xy * hS;
    vec2 deltaTC = P / numCapas;

    vec2 TCactuales = tc;
    float valorMapaProfActual = texture(MapaProf, TCactuales).r;

    while(capaActual < valorMapaProfActual) {

        TCactuales -= deltaTC;

        valorMapaProfActual = texture(MapaProf, TCactuales).r;

        capaActual += profCapa;
    }

    vec2 anteriorTC = TCactuales + deltaTC;

    float profPosterior  = valorMapaProfActual - capaActual;
    float profAnterior = texture(MapaProf, anteriorTC).r - capaActual + profCapa;


    float peso = profPosterior / (profPosterior - profAnterior);

    return (anteriorTC * peso + TCactuales * (1.0 - peso));
}
```

Figura 39: Shader de fragmentos, función para calcular el paralaje de una textura.

e) **calculaSombraNoCubo( )**: Función encargada de calcular si el fragmento está situado en alguna sombra del mapa de sombreado, empleará el algoritmo PCF para suavizar las diferencias de las zonas sombreadas y las que no.

```
float calculaSombraNoCubo(vec4 fp) {
    vec3 pc = fp.xyz / fp.w;
    pc = pc * 0.5 + 0.5;

    float profActual = pc.z;

    float bias = 0.005;
    float shadow = 0.0;
    vec2 TamTexel = 1.0 / textureSize(MapaSombra, 0);
    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            float profPCF = texture(MapaSombra, pc.xy + vec2(x, y) * TamTexel).r;
            shadow += profActual - bias > profPCF  ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

    if(pc.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

Figura 40: Shader de fragmentos, función para calcular las sombras a partir de una textura.

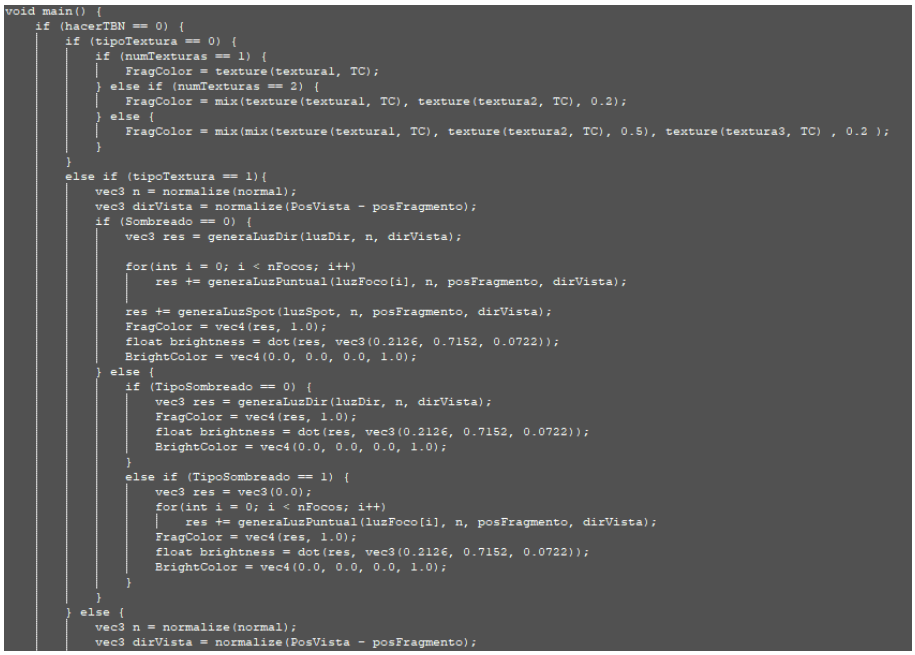
f) **calculaSombraCubo( )**: Función encargada de calcular la cantidad de sombra que va a recibir el fragmento según su posición en el cubo de sombras, emplearemos un algoritmo de PCF diferente para suavizar la sombra, el cual consistirá en usar un disco con 20 direcciones unitarias en 3D, en cada dirección del disco calcularemos si está cubierta por una sombra en el mapa cúbico y la media de los que toquen una sombra respecto a todas las direcciones observadas en total será la cantidad de sombra que recibirá nuestro fragmento, para no hacer muy oscura nuestra sombra en vez de dividir entre las 20 observaciones como sería lo lógico, dividiremos entre 21 para que nunca se llegue al valor 1.0 y aparezcan totalmente oscuras nuestras sombras.

```
float calculaSombraCubo(vec3 fp) {
    vec3 frag_luz = fp - luzFoco[0].posicion;
    float profActual = length(frag_luz);
    float shadow = 0.0;

    float bias = 0.15;
    int muestras = 20;
    float distancia_camara = length(vistaCamara - fp);
    float radio = (1.0 + (distancia_camara / planoLejano)) / 25.0;
    for(int i = 0; i < muestras; i++) {
        float profCercana = texture(CuboSombra, frag_luz + gridSamplingDisk[i] * radio).r;
        profCercana *= planoLejano;
        if(profActual - bias > profCercana)
            shadow += 1.0;
    }
    shadow = shadow / 21.0;
    return shadow;
}
```

Figura 41: Shader de fragmentos, función para calcular las sombras a partir de un cubo de sombras.

Ahora veremos como se utiliza todo lo que hemos visto para ejecutar nuestro shader y obtener el resultado final, el shader se encarga de calcular los tres tipos de luces utilizando los métodos explicados antes y suma los tres resultados, en caso de tratar con texturas del tipo 0 no tendremos en cuenta la luz, aunque no se mostrarán estos casos en la práctica. El resto de tipos de configuración si tratarán la luz recibida de este sumatorio que será lo que devuelva el shader. Cabe resaltar que si tratamos de una textura de tipo 1 con material que tiene más de una textura difusa tendremos que hacer los cálculos de las luces para cada una y posteriormente sumarlas para que se haga la mezcla de estas. En la salida de BrigthColor devolveremos todo oscuro ya que no queremos que la floración afecte a las figuras por el momento. Además en caso de que tengamos que calcular las sombras, este cálculo se hará en los métodos que calculan la luz, oscureciendo las componentes difusa y especular en caso de haber sombra.

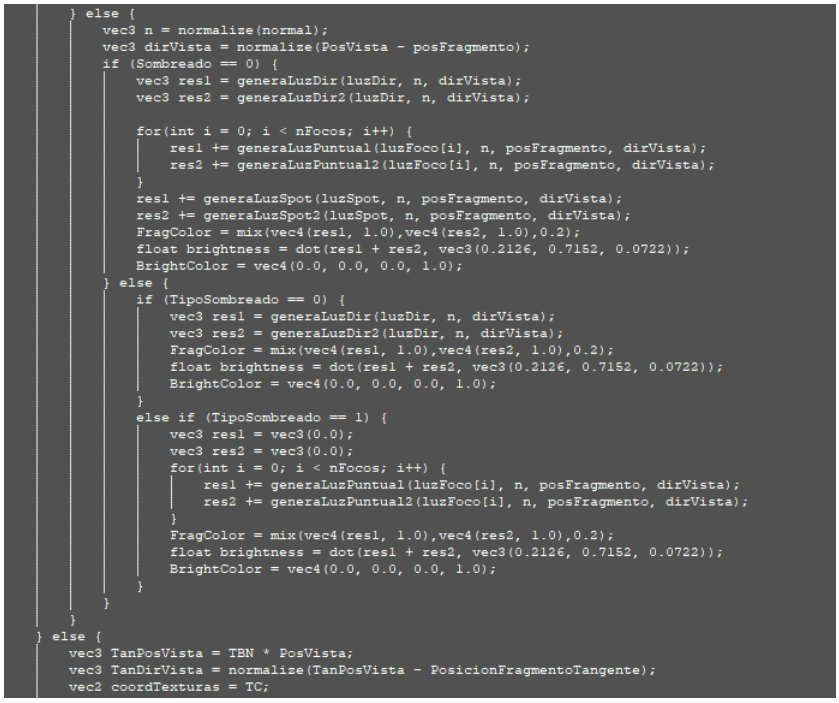


```
void main() {
    if (hacerTBN == 0) {
        if (tipoTextura == 0) {
            if (numTexturas == 1) {
                FragColor = texture(textura1, TC);
            } else if (numTexturas == 2) {
                FragColor = mix(texture(textura1, TC), texture(textura2, TC), 0.2);
            } else {
                FragColor = mix(mix(texture(textura1, TC), texture(textura2, TC), 0.5), texture(textura3, TC) , 0.2 );
            }
        }
        else if (tipoTextura == 1){
            vec3 n = normalize(normal);
            vec3 dirVista = normalize(PosVista - posFragmento);
            if (Sombreado == 0) {
                vec3 res = generaLuzDir(luzDir, n, dirVista);

                for(int i = 0; i < nFocos; i++)
                    res += generaLuzPuntual(luzFoco[i], n, posFragmento, dirVista);

                res += generaLuzSpot(luzSpot, n, posFragmento, dirVista);
                FragColor = vec4(res, 1.0);
                float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
                BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
            } else {
                if (TipoSombreado == 0) {
                    vec3 res = generaLuzDir(luzDir, n, dirVista);
                    FragColor = vec4(res, 1.0);
                    float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
                    BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
                }
                else if (TipoSombreado == 1) {
                    vec3 res = vec3(0.0);
                    for(int i = 0; i < nFocos; i++)
                        res += generaLuzPuntual(luzFoco[i], n, posFragmento, dirVista);
                    FragColor = vec4(res, 1.0);
                    float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
                    BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
                }
            }
        } else {
            vec3 n = normalize(normal);
            vec3 dirVista = normalize(PosVista - posFragmento);
```

Figura 42: Shader de fragmentos, fragmento del main.



```
        } else {
            vec3 n = normalize(normal);
            vec3 dirVista = normalize(PosVista - posFragmento);
            if (Sombreado == 0) {
                vec3 res1 = generaLuzDir(luzDir, n, dirVista);
                vec3 res2 = generaLuzDir2(luzDir, n, dirVista);

                for(int i = 0; i < nFocos; i++) {
                    res1 += generaLuzPuntual(luzFoco[i], n, posFragmento, dirVista);
                    res2 += generaLuzPuntual2(luzFoco[i], n, posFragmento, dirVista);
                }
                res1 += generaLuzSpot(luzSpot, n, posFragmento, dirVista);
                res2 += generaLuzSpot2(luzSpot, n, posFragmento, dirVista);
                FragColor = mix(vec4(res1, 1.0),vec4(res2, 1.0),0.2);
                float brightness = dot(res1 + res2, vec3(0.2126, 0.7152, 0.0722));
                BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
            } else {
                if (TipoSombreado == 0) {
                    vec3 res1 = generaLuzDir(luzDir, n, dirVista);
                    vec3 res2 = generaLuzDir2(luzDir, n, dirVista);
                    FragColor = mix(vec4(res1, 1.0),vec4(res2, 1.0),0.2);
                    float brightness = dot(res1 + res2, vec3(0.2126, 0.7152, 0.0722));
                    BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
                }
                else if (TipoSombreado == 1) {
                    vec3 res1 = vec3(0.0);
                    vec3 res2 = vec3(0.0);
                    for(int i = 0; i < nFocos; i++) {
                        res1 += generaLuzPuntual(luzFoco[i], n, posFragmento, dirVista);
                        res2 += generaLuzPuntual2(luzFoco[i], n, posFragmento, dirVista);
                    }
                    FragColor = mix(vec4(res1, 1.0),vec4(res2, 1.0),0.2);
                    float brightness = dot(res1 + res2, vec3(0.2126, 0.7152, 0.0722));
                    BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
                }
            }
        }
    } else {
        vec3 TanPosVista = TBN * PosVista;
        vec3 TanDirVista = normalize(TanPosVista - PosicionFragmentoTangente);
        vec2 coordTexturas = TC;
```

Figura 43: Shader de fragmentos, fragmento del main.

```
} else {
    vec3 TanPosVista = TBN * PosVista;
    vec3 TanDirVista = normalize(TanPosVista - PosicionFragmentoTangente);
    vec2 coordTexturas = TC;
    if (paralaje == 1) {
        coordTexturas = ParallaxMapping(TC,TanDirVista);
    }
    vec3 n = texture(MapaNormal, coordTexturas).rgb;
    n = normalize(n * 2.0 - 1.0);
    if (Sombreado == 0) {
        vec3 res = generaLuzDirTBN(luzDir, n, TanDirVista);
        for(int i = 0; i < nFocos; i++)
            res += generaLuzPuntualTBN(luzFoco[i], n, PosicionFragmentoTangente, TanDirVista);
        res += generaLuzSpotTBN(luzSpot, n, PosicionFragmentoTangente, TanDirVista);
        FragColor = vec4(res, 1.0);
        float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
        BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        if (TipoSombreado == 0) {
            vec3 res = generaLuzDirTBN(luzDir, n, TanDirVista);
            FragColor = vec4(res, 1.0);
            float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
            BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
        }
        else if (TipoSombreado == 1) {
            vec3 res = vec3(0.0);
            for(int i = 0; i < nFocos; i++)
                res += generaLuzPuntualTBN(luzFoco[i], n, PosicionFragmentoTangente, TanDirVista);
            FragColor = vec4(res, 1.0);
            float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
            BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
        }
    }
}
```

Figura 44: Shader de fragmentos, fragmento del main.

2. **ShaderAssimp**: Este shader es completamente similar al anterior pero con las siguientes diferencias, este shader va a ser tratado siempre como si hacerTBN fuese 1, por lo que hará en el shader de vértices las mismas operaciones que el de las figuras en el caso que utilicemos tangentes y binormales.

   Para el shader de vértices solo tenemos un ligero cambio en las texturas, usaremos unicamente dos texturas para calcular los colores difusos y otra para las normales. En este caso serán las uniforms $texture\_diffuse1$ y $texture\_normal1$, en el resto de métodos y operaciones utiliza exactamente las mismas.

   Podríamos haber utilizado el mismo shader que las figuras en Assimp ya que son parecidos, pero el hecho de que los modelos de Assimp sean tratados como objetos distintos que las figuras que diseñamos nosotros en la aplicación, me pareció razonable separarlos en shaders diferentes para en un futuro que estos tengan tratos diferentes si decido extender mi aplicación.

3. **ShaderCaja**: Este shader va a renderizar en nuestra pantalla el fondo de la escena, por lo que no tendrá muchas complejidades para ello. Es decir, solo dispone de shader de vértices y fragmentos. En los cuales el primero solo se encarga de recibir la posición y las coordenadas de textura, transforma las posiciones usando la matriz de proyección y la de vista, ya que no hay que modelar aquí, y envía los resultados al de fragmentos. Este solo tiene que recibir las coordenadas de textura y pintar el fragmento según indique la textura cúbica que le mandemos mediante la uniform $mapaCaja$.

4. **ShaderBlur**: Para poder difuminar una textura usando este shader tenemos que pasarle un rectángulo 2D que ocupe toda la pantalla para dibujar el resultado allí, por lo que su shader de vértices solamente se tendrá que encargar de recibir su posición y mandársela al shader de fragmentos tal cual la recibe, al igual que las coordenadas de textura.

   Ahora una vez en el shader de fragmentos tendremos que aplicar el algoritmo de difuminación explicado en la sección 3.6, por lo que necesitaremos de tres uniforms para poder llevarlo a cabo. La primera es la textura que queramos difuminar $imagenParaModificar$, la segunda es una uniform que nos indique si queremos empezar a hacer el mapeo de forma horizontal o vertical y la tercera el array de pesos gaussianos que vamos a usar en el algoritmo. Una vez conozcamos estas uniforms es ejecutar el algoritmo y obtener la imagen resultante difuminada.

```
out vec4 FragColor;

in vec2 TC;

uniform sampler2D imagenParaModificar;

uniform bool horizontal;
uniform float pesos[5] = float[] (0.2270270270, 0.1945945946, 0.1216216216, 0.0540540541, 0.0162162162);

void main() {
    vec2 tOff = 1.0 / textureSize(imagenParaModificar, 0);
    vec3 res = texture(imagenParaModificar, TC).rgb * pesos[0];
    if(horizontal) {
        for(int i = 1; i < 5; i++) {
            res += texture(imagenParaModificar, TC + vec2(tOff.x * i, 0.0)).rgb * pesos[i];
            res += texture(imagenParaModificar, TC - vec2(tOff.x * i, 0.0)).rgb * pesos[i];
        }
    }
    else {
        for(int i = 1; i < 5; i++) {
            res += texture(imagenParaModificar, TC + vec2(0.0, tOff.y * i)).rgb * pesos[i];
            res += texture(imagenParaModificar, TC - vec2(0.0, tOff.y * i)).rgb * pesos[i];
        }
    }
    FragColor = vec4(res, 1.0);
}
```

Figura 45: Shader de fragmentos que se encargará de difuminar la escena de luces.

5. **ShaderSombra**: Este shader solo va a ejecutar los objetos de la escena para guardar en una textura sus profundidades haciendo uso del FBO creado en la clase Sombra, por lo que su shader de vértices tendrá que disponer de una entrada para las posiciones de los vértices, además de contar con dos uniforms *model* y $lightSpaceMatrix$, la primera va a modelar el objeto en la escena y la segunda lo va a mover para que se sitúe en el espacio con la luz como punto de referencia y proyectado. Una vez terminada la transformación se le envía a un shader de fragmentos que no realiza ninguna operación ya que no lo necesita, al tener que el FBO automáticamente detectará las profundidades en nuestro espacio y las pintará en su textura, lo que hace que este shader de fragmentos no tenga que hacer nada.

6. **ShaderSombraCubo**: Este shader va a disponer de un shader geométrico para poder distribuir en cada cara del cubo la sombra del objeto que le corresponde almacenar.

Su shader de vértices es mucho más simple que el de las sombras direccionales, la entrada será la misma, las posiciones de vértices, etc. Pero esta vez solo dispondrá de la uniform *model* para poder mover los vértices al espacio global. Ahora entramos al shader geométrico donde tendremos que situar los puntos que nos lleguen del shader de vértices y proyectar primitivas en cada cara del cubo, emitiendo 6 figuras, una para cada cara. Para ello necesitamos un array de 6 matrices para proyectar en cada una de ellas, que cargaremos en el shader con la uniform $sombraMatriz[]$, donde cada matriz tendrá una conversión a espacio vista y proyección perspectiva apropiados según la cara del cubo que le corresponda. El shader de fragmentos solo tendrá que almacenar en el buffer de profundidad de cada cara del cubo los valores de profundidad que le corresponda al fragmento, $gl\_FragDepth$, calculando la distancia del fragmento al punto de emisión de la luz y dividiendo el resultado entre el valor del plano lejano de la proyección perspectiva para situarlo en el rango entre 0 y 1. Al igual que en el Shader anterior, el FBO creado en la clase Sombra pintará en cada cara del cubo el valor de profundidad correspondiente, generando la textura cúbica de sombras de la escena.

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 sombraMatriz[6];

out vec4 posFragmento;

void main() {
    for(int c = 0; c < 6; c++) {
        gl_Layer = c;
        for(int i = 0; i < 3; i++) {
            posFragmento = gl_in[i].gl_Position;
            gl_Position = sombraMatriz[c] * posFragmento;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Figura 46: Shader geométrico en ShaderSombraCubo.

```
#version 330 core
in vec4 posFragmento;

uniform vec3 posLuz;
uniform float planoLejano;

void main()
{
    gl_FragDepth = length(posFragmento.xyz - posLuz) / planoLejano;
}
```

Figura 47: Shader de fragmentos en ShaderSombraCubo.

7. **ShaderFocosLuz**: Este shader es el encargado de dibujar los focos de las luces puntuales, su funcionamiento es sencillo, recibe la posición de un cubo junto a las uniforms para modelarlo y proyectarlo. Una vez ubicado se le envía al shader de fragmentos que

se va a encargar de pintarlo de blanco y devolver el color del fragmento, pero también debemos recordar que envíe el color blanco por el segundo canal de salida del FBO para que cuando usemos la técnica de la floración podamos tratar las texturas con brillos.

```
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;

void main() {
    vec3 res = vec3(15.0, 15.0, 15.0);
    FragColor = vec4(res, 1.0);
    float brightness = dot(res, vec3(0.2126, 0.7152, 0.0722));
    if(brightness > 1.0)
    |   BrightColor = vec4(res, 1.0);
    else
    |   BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Figura 48: Shader de fragmentos encargado de pintar los focos.

8. **ShaderBlooming**: Una vez ejecutado el shaderBlur y generado la textura difusa deseada, este shader se tiene que encargar de mezclar esta textura resultante con la textura original. Al igual que el shader de difuminación, tiene que recibir un rectángulo 2D que ocupe toda la pantalla y pintar el resultado sobre éste, luego el shader de vértices solo va a recibir las posiciones de este rectángulo y las coordenadas de textura, éste se las pasará directamente al shader de fragmentos.

   En el shader de fragmentos tenemos que tener en cuenta una cosa, necesitamos dos texturas para poder mezclarlas y generar la imagen definitiva, luego las cargaremos en las uniforms *escena* y *bluringCompleto*. Una vez tengamos las texturas solo deberemos de sumarlas, aplicarles HDR para que se diferencien los brillos y aplicarles un poco de corrección gamma para corregir los brillos de pantalla. Una vez hecho esto se devolverá la imagen definitiva que se renderizará en pantalla.

```
out vec4 FragColor;

in vec2 TC;

uniform sampler2D escena;
uniform sampler2D bluringCompleto;

void main() {
    const float gamma = 1.4;
    vec3 hdrColor = texture(escena, TC).rgb;
    vec3 bloomColor = texture(bluringCompleto, TC).rgb;
    |
    hdrColor += bloomColor;

    vec3 result = vec3(1.0) - exp(-hdrColor * 1.0);

    // correccion gamma
    result = pow(result, vec3(1.0 / gamma));
    FragColor = vec4(result, 1.0);
}
```

Figura 49: Shader de fragmentos encargado de generar la imagen final con floración de luz.

9. **ShaderParticulas**: Este es uno de los shaders que va a disponer de un shader geométrico, además de recibir las entradas del VAO distintas a como las reciben los demás. Empezamos hablando del shader de vértices que como bien hemos dicho en la sección 3.7, recibirá del VAO la posición inicial, velocidad y tiempo de vida de una partícula. Necesitará de una serie de uniforms para poder realizar los cálculos de la nueva posición de la partícula, estos son: *tiempo* que contendrá el tiempo actual, *acc* para la aceleración de la partícula y *t_ini* que llevará el tiempo de nacimiento de la partícula. Con estas uniforms y los parámetros de entrada se calculará la nueva posición con la siguiente fórmula:

$$pos = pos0 + vel0 * (tiempo - t\_ini) + acc * (tiempo - t\_ini)^2 \quad (3)$$

Al calcular la nueva posición se comprobará que el tiempo que lleva la partícula en la escena no exceda el tiempo de vida de ésta. Si lo hace configuraremos su alpha para que sea cada vez más transparente para que la desaparición no sea de golpe.

```
uniform float tiempo;
uniform float t_ini;
uniform vec3 acc;

out Particle
{
    vec3 position;
    float alpha;
} particle;

void main()
{
    float t = tiempo - t_ini;
    particle.position = aPos + aVel * t + acc * (t * t * 0.5);

    if (1.0 < (t - vida) * -2.0)
    |   particle.alpha = 1.0;
    else
    |   particle.alpha = (t - vida) * -2.0;
}
```

Figura 50: Shader de vértices encargado de calcular la posición y la transparencia de las partículas.

Ahora vamos con el shader geométrico que efectuará la parte más importante del sistema de partículas, trataremos de crear la primitiva de un cuadrado (triangleStrip) que rodee la posición de nuestra partícula, además tendremos que encargarnos que este cuadrado nos mire siempre hacia la cámara para conseguir la sensación de 3D. De ahí que necesitemos las uniforms fundamentales que son *model*, *view* y *projection*, con esto generaremos nuestro cuadrado de la siguiente manera:

Primero cogemos la posición de la partícula y la pasamos al espacio de vista, es decir lo pasamos por la matriz modelo y por la de vista, por la proyección aún no. Después calculamos los vértices del cuadrado que lo rodea, multiplicamos a estos por la matriz de proyección y los emitimos, cuando hemos tratado los 4 vértices, emitimos la primitiva y ya hemos terminado.

```
void main() {

    if (particle[0].alpha <= 0)
        return;

    fragment.alpha = particle[0].alpha;

    vec4 center = view * model * vec4(particle[0].position, 1);


    vec2 uv = vec2(1, 1);
    vec4 p = center;
    p.xy += uv;
    fragment.uv = uv;
    gl_Position = projection * p;
    EmitVertex();

    uv = vec2(-1, 1);
    p = center;
    p.xy += uv;
    fragment.uv = uv;
    gl_Position = projection * p;
    EmitVertex();

    uv = vec2(1, -1);
    p = center;
    p.xy += uv;
    fragment.uv = uv;
    gl_Position = projection * p;
    EmitVertex();

    uv = vec2(-1, -1);
    p = center;
    p.xy += uv;
    fragment.uv = uv;
    gl_Position = projection * p;
    EmitVertex();

    EndPrimitive();
}
```

Figura 51: Shader geométrico encargado de calcular el cuadrilátero que rodea la posición de la partícula y gira con nuestra cámara.

Ahora que ya hemos generado nuestra primitiva, toca la parte del shader de fragmentos, tenemos que calcular su transparencia y para ello usamos una textura circular, es decir, descartamos los fragmentos que no estén en la circunferencia contenida en el cuadrado, cargamos las coordenadas de la textura de las partículas, que se ubican en la uniform *textura*, y posteriormente hacemos que los fragmentos más alejados del centro del circulo sean más transparentes que los que están en el centro. Una vez hecho esto se pintará la partícula correctamente y habremos terminado el proceso de renderización de esta.

```
in Fragment {
    vec2 uv;
    float alpha;
} fragment;

out vec4 fragColor;

uniform sampler2D textura;

void main() {
    float d = dot(fragment.uv, fragment.uv);

    if(d > 1.0)
        discard;

    float a = 1-d;

    vec3 res = texture(textura, fragment.uv).rgb;
    if (res.r <= 0.5 && res.g <= 0.5 && res.b <= 0.5) {
        discard;
    }

    fragColor = vec4(res, a * fragment.alpha * 0.5);
}
```

Figura 52: Shader de fragmentos encargado de pintar las partículas.

4.3. Resultados obtenidos tras la ejecución de nuestro programa

Para la obtención de los resultados hemos decidido hacer tres pruebas con un escenario diferente cada una. El primer escenario no tendrá sombras, solo usaremos la iluminación para así poder probar los focos y las técnicas de floración de la luz. Hemos cargado una superficie de ladrillos usando la técnica de paralaje para darle un toque más realista. Posteriormente colocaremos tres focos de luz puntual y distintos cubos con diferentes tipos de texturas cada uno, incluido algunos con mapeo normal y por paralaje. Además introduciremos dos modelos de Assimp y los ubicaremos en la escena, el resto será configurar las luces spot y direccional. Tendremos que observar la escena con y sin el modelo de Blinn-Phong para poder comprobar la diferencia y además probar la escena con y sin floración para ver los cambios que este efecto desemboca.

Para la segunda escena vamos a usar sombras del primer tipo, es decir solo con iluminación de luz direccional. Trataremos de dibujar cubos y modelos de forma estratégica para ver que efectivamente van a proyectar sombras y los que sean cubiertos por alguna de estas notarán esas zonas oscuras. También deberemos de probar con y sin Blinn-Phong para ver las diferencias. La floración no hará falta en esta prueba ya que no dispondremos de focos de luz para probarla.

La última prueba será probar las sombras de segundo tipo que serán las que genere la luz posicional. Dibujaremos la misma escena que en la anterior prueba y haremos el mismo procedimiento. Los resultados deberán de ser similares.

Una vez realizadas las pruebas hemos podido comprobar que los resultados son correctos y los podemos observar en las siguientes imágenes:

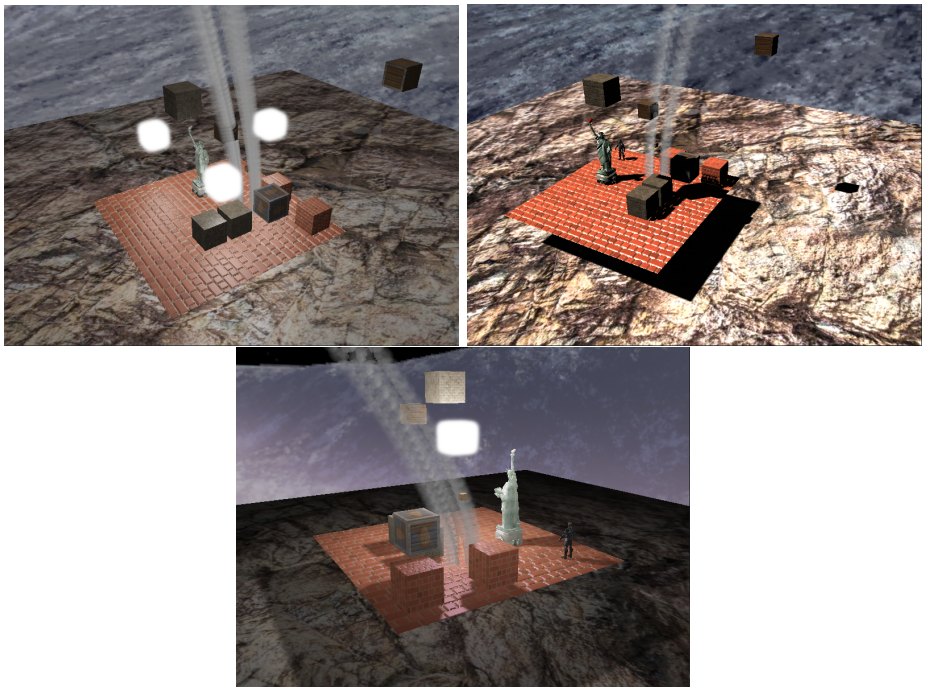


Figura 53: Ejecución de las tres pruebas del motor.

5. Conclusiones finales

Para concluir este proyecto, vamos a hacer un inciso analizando los distintos aspectos que se han visto en éste. En primer lugar, tenemos una gran parte de investigación sin ninguna base previa que personalmente, me ha servido para coger experiencia en futuras situaciones. Además, de cómo organizar un plan de trabajo y poder llevarlo a la perfección cumpliendo las fechas previstas.

Por otra parte, he conseguido poner en práctica muchos conocimientos aprendidos durante todo el plan de estudios como por ejemplo el Álgebra Lineal y la Programación Orientada a Objetos. Gracias a esto, he conseguido comprender de manera correcta la progrmación gráfica con shaders y los motivos por los que se utilizan. Por otra parte, la Programación Orientada a Objetos, me ha servido como base para poder desarrollar el programa, ya que me ha permitido abstraer todos los conceptos de una escena, dando como resultado las clases de todos sus componentes y combinándolas para generar el resultado final.

Para finalizar, he de resaltar la gran dificultad a la hora de realizar las pruebas finales del motor gráfico, como por ejemplo: las sombras, los sistemas de partículas o el blooming con HDR. Estas dificultades, han sido subsanadas gracias a un fuerte trabajo de investigación y esfuerzo, por el cual he conseguido resolverlas de manera correcta, para acabar la escena con éxito, y entregar el proyecto a tiempo y funcionando.

Referencias

[0] Wikipedia OpenGl,
https://es.wikipedia.org/wiki/OpenGL

[1] Página oficial de GLFW,
https://www.glfw.org

[2] Página oficial de khronos, cargadores de librerias openGL,
https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

[3] Página oficial de khronos, GLSL,
https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)

[4] Página oficial de GLM,
https://glm.g-truc.net/0.9.9/index.html

[5] Página oficial de khronos, VAO, VBO y shaders
https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL)

[6] LearnOpenGL, Texturas,
https://learnopengl.com/Getting-started/Textures

[7] Página oficial de khronos, Shader de vértices,
https://www.khronos.org/opengl/wiki/Vertex_Shader

[8] Página oficial de khronos, Shader de fragmentos,
https://www.khronos.org/opengl/wiki/Fragment_Shader

[9] LearnOpenGL, Shader Geométrico,
https://learnopengl.com/Advanced-OpenGL/Geometry-Shader

[10] LearnOpenGL, Clase Mesh,
https://learnopengl.com/Model-Loading/Mesh

[11] LearnOpenGL, Clase Model,
https://learnopengl.com/Model-Loading/Model

[12] LearnOpenGL, Assimp,
https://learnopengl.com/Model-Loading/Assimp

[13] Página oficial de Assimp,
http://assimp.sourceforge.net/lib_html/index.html

[14] Agueda Mata y Miguel Reyes, Dpto. de Matemática Aplicada, FI-UPM,
http://www.dma.fi.upm.es/personal/mreyes/Algebra/Teoria/AL_ap_04.pdf

[15] Apuntes de la asignatura Informática Gráfica en la Universidad de Oviedo, Tema 2, Transformaciones lineales en 3D,
http://di002.edv.uniovi.es/~rr/Tema2.pdf

[16] LearnOpenGL, Aplicaciones Lineales, Rotación,
https://learnopengl.com/Getting-started/Transformations

[17] Artículo sobre proyecciones en openGL, songho.ca,
http://www.songho.ca/opengl/gl_projectionmatrix.html

[18] LearnOpenGL, Camera, view space,
https://learnopengl.com/Getting-started/Camera

[19] LearnOpenGL, Sistemas de coordenadas en 3D,
https://learnopengl.com/Getting-started/Coordinate-Systems

[20] LearnOpenGL, Fondo de una escena (skybox),
https://learnopengl.com/Advanced-OpenGL/Cubemaps

[21] LearnOpenGL, Componentes de la luz,
https://learnopengl.com/Lighting/Basic-Lighting

[22] LearnOpenGL, Iluminación aplicada a materiales,
https://learnopengl.com/Lighting/Materials

[23] LearnOpenGL, Tipos de luz según la fuente de emisión,
https://learnopengl.com/Lighting/Light-casters

[24] LearnOpenGL, Modelo de Blinn-Phong,
https://learnopengl.com/Advanced-Lighting/Advanced-Lighting

[25] LearnOpenGL, Corrección Gamma,
https://learnopengl.com/Advanced-Lighting/Gamma-Correction

[26] LearnOpenGL, Mapeado de Sombras,
https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

[27] LearnOpenGL, Mapeado de Sombras con luz multidireccional,
https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

[28] LearnOpenGL, Mapeado de Normales,
https://learnopengl.com/Advanced-Lighting/Normal-Mapping

[29] LearnOpenGL, Mapeado por paralaje,
https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

[30] LearnOpenGl, Zonas de alto rango de iluminación, HDR,
https://learnopengl.com/Advanced-Lighting/HDR

[31] LearnOpenGL, Floración de la luz,
https://learnopengl.com/Advanced-Lighting/Bloom

[32] LearnOpenGL, FrameBuffers,
https://learnopengl.com/Advanced-OpenGL/Framebuffers

[33] Microsoft, gráficos y atenuación de la luz,
https://docs.microsoft.com/es-es/windows/uwp/graphics-concepts/attenuation-and-spotlight-factor

[34] GameDev, sistemas de particulas con shader geométrico,
https://www.genericgamedev.com/effects/parametric-gpu-accelerated-particles/

[35] OpenGl-Tutorial.org, sistemas de particulas implementación,
http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

[36] Algoritmo para calcular tangentes y binormales,
https://gamedev.stackexchange.com/questions/68612/how-to-compute-tangent-and-bitangent-vectors

[37] Código original clase Model.h,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/model.h

[38] Código original de la clase Mesh.h,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/mesh.h

[39] Código original de la clase Camera.h
https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.4.camera_class/camera_class.cpp

[40] Código donde aparece el método para cargar el cubo,
https://learnopengl.com/code_viewer_gh.php?code=src/4.advanced_opengl/6.1.cubemaps_skybox/cubemaps_skybox.cpp

[41] Código donde aparece el método para cargar las texturas,
https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/4.2.lighting_maps_specular_map/lighting_maps_specular.cpp